Factor Graphs and Algorithms

Brendan J. Frey
The Beckman Institute
405 North Mathews Avenue
Urbana, IL 61801, USA
frey@cs.utoronto.ca

Frank R. Kschischang
ECE Department
University of Toronto
Toronto, Ontario M5S 3G4, CANADA
frank@comm.utoronto.ca

Hans-Andrea Loeliger
Endora Tech AG
Gartenstraße 120
CH-4052 Basel, SWITZERLAND
haloeliger@access.ch

Niclas Wiberg
Dept. of Electrical Engineering
Linköping University
S-581 83 Linköping, SWEDEN
nicwi@isy.liu.se

Abstract—A factor graph is a bipartite graph that expresses how a global function of several variables factors into a product of local functions. Factor graphs subsume many other graphical models, including Bayesian networks, Markov random fields, and Tanner graphs. We describe a general algorithm for computing "marginals" of the global function by distributed message-passing in the corresponding factor graph. A wide variety of algorithms developed in the artificial intelligence, statistics, signal processing, and digital communications communities can be derived as specific instances of this general algorithm, including Pearl's "belief propagation" and "belief revision" algorithms, the fast Fourier transform, the Viterbi algorithm, the forward/backward algorithm, and the iterative "turbo" decoding algorithm.

1 Introduction

A *factor graph* is a bipartite graph that expresses how a global function of several variables factors into a product of local functions.

Example 1. Suppose the real-valued function $g(x_1, x_2, \ldots, x_5)$ of five variables can be written as

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1, x_2) f_B(x_2, x_3, x_4) f_C(x_4, x_5) f_2(x_2) f_4(x_4).$$

We refer to g as the *global function* and f_A, f_B, f_C, f_2 and f_4 as *local functions*. The set of arguments of each local function is a subset of the arguments of g.

This factorization can be expressed via the factor graphs shown in Fig. 1. A factor graph consists of two types of vertices: those associated with variables (the unfilled circles in Fig. 1, called *variable nodes*) and those associated with local functions (the filled circles in Fig. 1, called *subset nodes*). The edges of the factor graph are precisely those that join the variable node for x_i to the subset node for f if and only if x_i is an argument of f.

In general, let $X = \{x_i\}_{i \in N}$ be a collection of variables, indexed by a finite set $N = \{1, 2, 3, \ldots, n\}$. If E is a nonempty subset of N, we denote by X_E the subset of X

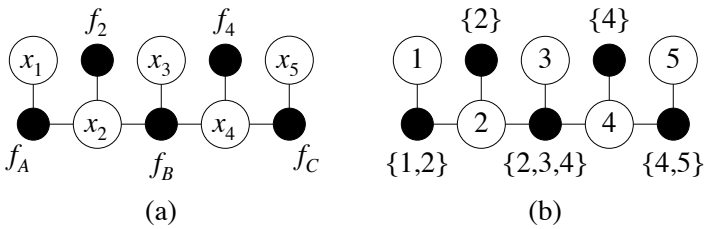

Figure 1: A factor graph that expresses that a global function factors as the product of local functions $f_A(x_1, x_2)f_B(x_2, x_3, x_4)f_C(x_4, x_5)f_2(x_2)f_4(x_4)$. (a) Variable/function form, (b) index/subset form.

indexed by E. For each $i \in N$ the variable x_i takes on values from the *alphabet* A_i. In this paper, we will take A_i to be finite for all $i \in N$, though this assumption is not essential. A particular assignment of a value to each of the variables of X will be referred to as a *configuration* of the variables. Configurations of the variables can be viewed as being elements of the Cartesian product $W = \prod_{i \in N} A_i$, called the *configuration space*. An element $w = (w_1, w_2, \ldots, w_n) \in W$, with $w_i \in A_i$ is equivalent to the variable assignment $x_1 = w_1$, $x_2 = w_2$, $\ldots$, $x_n = w_n$, and vice versa. We will have occasion to view configurations both as assignments of values to variables, and as elements of W.

We are interested in functions with the elements of X as arguments, i.e., functions with domain W. Let $g : W \to R$ denote such a function, referred to here as the *global function*. For the moment we take the codomain, R, of g to be the set of real numbers, though later, following [1–4], we shall allow R to be any semiring.

Let Q be a collection of subsets of N (i.e., a subset of the power set of N), not including the empty set. Suppose that g can be written as a product of local functions with arguments indexed by the elements of Q, i.e.,

$$g(X) = \prod_{E \in Q} f_E(X_E). \tag{1}$$

Then a factor graph representation of (1) is a bipartite graph with vertex set $N \cup Q$, and edge set $\{\{i, E\} : i \in N, E \in Q, i \in E\}$. As stated earlier, we refer to those vertices that are elements of N as *variable nodes* and those vertices that are elements of Q as *subset nodes*. An edge joins a variable node i to a subset node E if and only if $i \in E$, hence the factor graph is a graphical representation of the relation "element of" in $N \times E$. In Example 1, we have $N = \{1, 2, 3, 4, 5\}$, and $Q = \{\{1, 2\}, \{2, 3, 4\}, \{4, 5\}, \{2\}, \{4\}\}$.

2 Examples of Factor Graphs

2.1 Set Membership Indicator Functions

In coding theory, as in systems theory, one is often interested in describing *subsets* B of the set W of possible configurations. In the coding context, such a subset defines a *code*. while in systems theory, B is referred to as the system's *behavior*, and each element of B is a *valid* configuration.

Set membership can in general be described with a binary-valued global indicator func-

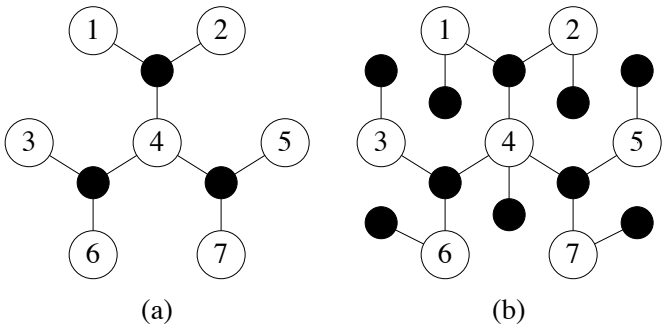

Figure 2: Factor graph representation of (a) a binary code, and (b) the *a posteriori* joint probability mass function of the codeword symbols.

tion. For a set $B \subset W$, define $I_B : W \to \{0, 1\}$ by

$$I_B(w) = \begin{cases} 1 & \text{if } w \in B; \\ 0 & \text{if } w \notin B. \end{cases}$$

It is often the case—by circumstance or by design—that this global indicator function factors into a product of *local* indicator functions, each of which checks for locally valid behavior. Specifically, it may happen that $g(X)$ factors as in (1), where each factor $f_E(X_E)$ of $g(X)$ is itself a binary-valued indicator function. A configuration $w \in W$ is said to be locally valid at E if the local function $f_E(w_E)$ evaluates to unity at the restriction w_E of w to E. Thus a configuration $w \in W$ is a valid configuration if and only if it is locally valid at all $E \in Q$.

For example, every binary linear block code can be described by a set of parity-check equations, in which each equation imposes the condition on a codeword $x = (x_1, x_2, \ldots, x_n)$ that its restriction x_E to a subset E of symbol positions must have even parity; i.e., summing in GF(2), $\sum_{i \in E} x_i = 0$.

Example 2. The factor graph corresponding to the binary code with parity-check matrix

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

is shown in Fig. 2(a).

More generally, one could impose more complicated constraints, for example requiring that x_E be an element of some more complicated linear code. Such codes were considered by Tanner [5]. Even more generally, following Wiberg, *et al.* [3, 6], one could introduce (unobserved) state variables, not considered part of the "codeword," but which participate in defining locally valid behavior. These "local check" descriptions are naturally described using a bipartite factor graph (called a Tanner graph in [3, 6] or, to emphasize that unobserved state variables are permitted, a Tanner-Wiberg-Loeliger graph [7]).

2.2 Indicator Functions and a Posteriori Probabilities

Continuing with Example 2, let us select, with uniform probability, a codeword $(x_1, \ldots, x_n)$ to transmit over a memoryless channel, and suppose that $y = (y_1, \ldots, y_n)$ is the corresponding observed channel output. The *a posteriori* joint probability distribution of

$\{x_1, \ldots, x_n\}$ is then linearly proportional to

$$f(x_1, x_2, \ldots, x_n) = \prod_{E \in Q} f_E(x_E) \prod_{i=1}^{n} f(y_i|x_i), \tag{2}$$

where, for each value of x_i, $f(y_i|x_i)$ is the corresponding likelihood function evaluated at the observed channel output. We observe that (2) admits a factor graph representation obtained by augmenting the factor graph representation of the code itself with "singleton" local functions $f(y_i|x_i)$ connected by a single edge to each variable. This is shown for our example code in Fig. 2(b).

As we shall see, in computations involving the factor graph, such "singleton" local function nodes can be absorbed into the corresponding variable node, so that effectively, the factor graph describing the *a posteriori* joint probability distribution, given the observed channel output, is equivalent to the factor graph of the code itself.

2.3 Markov Random Fields

A Markov random field (see, e.g., [8]) is a graphical model based on an undirected graph $G = (V, E)$ in which each vertex corresponds to a random variable. Denote by $n(v)$ the neighbors of $v \in V$, i.e., the set of vertices of V connected to v by a single edge of E. The graph G is a *Markov random field* (MRF) if the distribution $p(v_1, \ldots, v_n)$ satisfies the local Markov property: $(\forall v \in V)\ \ p(v|V \backslash \{v\}) = p(v|n(v))$. 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., [8–11]). See [12] for a brief discussion of the use of MRFs to describe codes.

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 in terms of a collection of Gibbs potential functions, defined on the set Q of maximal cliques in the MRF. (A clique is a collection of vertices which are all pairwise neighbors, and such a clique is maximal if it is not properly contained in any other clique.) In other words, the distribution factors as

$$p(v_1, v_2, \ldots, v_N) = Z^{-1} \prod_{E \in Q} \psi_E(V_E) \tag{3}$$

where Z^{-1} is a normalizing constant. For example (cf. Example 1), the MRF in Fig. 3(a) can be used to express the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = Z^{-1} \psi_A(v_1, v_2) \psi_B(v_2, v_3, v_4) \psi(v_4, v_5).$$

Clearly (3) 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 (3), 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 [12].)

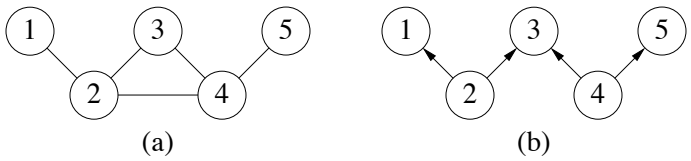

Figure 3: Other graphical models: (a) a Markov random field, and (b) a Bayesian network.

2.4 Bayesian Networks

Bayesian networks (see, e.g., [13–16]) 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" [13] have become an important tool in expert systems over the past decade. The first to connect Bayesian networks and belief propagation with applications in coding theory were MacKay and Neal [17], who independently re-discovered Gallager's earlier work on low-density parity-check codes [18] (including Gallager's decoding algorithm.) More recently, at least two papers [12, 19] develop a view of the "turbo decoding" algorithm [20] as an instance of probability propagation in a Bayesian network code model.

Each vertex 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), the distribution represented by the Bayesian network assumes the form

$$p(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} p(v_i|\mathbf{a}(v_i)), \tag{4}$$

where, if $\mathbf{a}(v_i) = \varnothing$, (i.e., v_i has no parents) then we take $p(v_i|\varnothing) = p(v_i)$. For example (cf. Example 1), Fig. 3(b) shows a Bayesian network that expresses the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = p(v_1|v_2)p(v_2)p(v_3|v_2, v_4)p(v_4)p(v_5|v_4).$$

Again, as in the Markov random field case, this graphical model expresses a factorization that is suitable for a factor graph representation.

3 The Sum-Product Algorithm

As in the previous section, let $g(X)$ be a global function over the variables in the set $X = \{x_i : i \in N\}$, with variable x_i taking on values in the finite set A_i. For the moment, we again take g as being real-valued.

In this section we describe a general algorithm that can be used to compute the *marginal functions*

$$G_i(x_i) \triangleq \sum_{x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1}, x_{i+1} \in A_{i+1}, \ldots, x_n \in A_n} g(x_1, \ldots, x_n) \tag{5}$$

for all variables x_i, $i \in N$. We adopt the convention that $\sum_{x_i} f(x_i) = \sum_{x_i \in A_i} f(x_i)$. Similarly, for a subset $J \subset N$, the notation $\sum_{x_i : i \in J} f(X_J)$ means that we sum over all possible configurations of the variables indexed by J. Thus $G_i(x_i) = \sum_{x_j : j \in N \setminus \{i\}} g(X)$.

The definition of marginal function can be extended an arbitrary subset J of N by defining

$$G_J(X_J) \triangleq \sum_{x_i : i \in N \setminus J} g(X).$$

If $g(x_1, \ldots, x_n)$ is a probability distribution, then $G_i(x_i)$ is a marginal distribution, and $G_J(X_J)$ is the joint probability distribution of the variables indexed by J.

When n, the number of arguments of g is small, we will sometimes use a modified notation for the marginal functions. We replace an argument x_i of g with a '+' sign to indicate that the corresponding variable is to be summed over, i.e., "marginalized out."

To perform marginalization, we take advantage of the factorization (1) of the global function, as represented by a factor graph. We use two basic properties of the factor graph: (1) products of local functions can be "gathered" along paths in the graph and (2) variables can be represented in "summary" outside of regions of the graph in which the variable is not "involved."

3.1 An Example

Example 3. To explain what we mean by the rather vague terminology of the previous sentence, let us consider the specific case in which the global function has the structure

$$g(x_1, x_2, x_3) = a(x_1) b(x_1, x_2) c(x_2, x_3). \tag{6}$$

This structure arises, for example, when $g(x)$ is the joint probability distribution of three random variables X_1, X_2, X_3 that form the Markov chain $X_1 \to X_2 \to X_3$ given some observation y.

Consider the computation of $g(x_1, +, +) = p(x_1|y)$. We write

$$\begin{aligned} g(x_1, +, +) &= \sum_{x_2} \sum_{x_3} a(x_1) b(x_1, x_2) c(x_2, x_3) \\ &= a(x_1) \underbrace{\sum_{x_2} b(x_1, x_2) \underbrace{\sum_{x_3} c(x_2, x_3)}_{c(x_2, +)}}_{bc(x_1, +, +)}, \end{aligned} \tag{7}$$

where we write $bc(x_1, x_2, x_3)$ for the product $b(x_1, x_2) c(x_2, x_3)$. In (7) we have identified the various factors that need to be computed to obtain $g(x_1, +, +)$. Our primary observation is that $g(x_1, +, +)$ can be computed knowing just $a(x_1)$ and $bc(x_1, +, +)$. The latter factor can be computed knowing just $b(x_1, x_2)$ and $c(x_2, +)$. These products can be "gathered" along the path from variable node x_3 to the variable node x_1 shown in Fig. 4. Note that variable x_3 can be "summarized," i.e., marginalized out, outside of the dashed region labeled 'x_3;' likewise for x_2.

Analyzing the computation of the remaining marginal functions in the same manner, we find that

$$g(+, x_2, +) = c(x_2, +) ab(+, x_2) \tag{8}$$

$$g(+, +, x_3) = \sum_{x_2 \in S_2} ab(+, x_2) c(x_2, x_3) = abc(+, +, x_3). \tag{9}$$

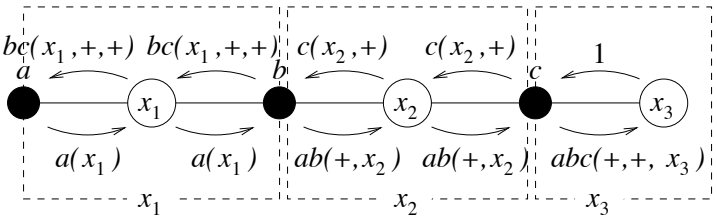

Figure 4: Distributed marginalization for a Markov chain example. For each i, the region in which x_i must be carried is indicated by a dashed box.

Comparing with Fig. 4, we observe that the various factors needed to compute each marginal function can be obtained as products of the "messages" sent along a particular path in the factor graph. The product of the messages carried in the two directions over any given edge is then a marginal function.

3.2 The General Algorithm

The sum-product algorithm operates via a "message passing" procedure that gathers local function products along paths in the factor graph. We assume that this graph is a tree, i.e., that it contains no cycles. The description of the algorithm is simplified by making the assumption that each node of the factor graph is a processor capable of transmitting and receiving "messages" along the edges to which it is connected.

In its simplest realization, the sum-product algorithm operates as follows. The basic operation at each node is to compute the product of the incoming messages at that node. At a subset node, this product is also multiplied by the local function associated with that node. These products are then transmitted on the "outgoing" edges, with the caveat that outgoing messages transmitted along a particular edge should contain no factors received on that edge. In this way, provided that the factor graph contains no cycles, the product of the message *transmitted* along an edge with the message *received* along that edge contains *all* factors of the global function.

This message-passing procedure is initiated at the leaf nodes in the factor-graph, i.e., those nodes on which only a single edge is incident. At leaf nodes that are subset nodes, the transmitted message is a representation of the local function associated with that node. At leaf nodes that are variable nodes, the transmitted message is a "unit" function (the transmission of which, in a practical realization, is of course unnecessary). All other nodes in the graph await the reception of enough messages to be able to compute an outgoing message. Effectively, these nodes wait until messages arrive on all but one edge; when this occurs, the incoming messages are multiplied together with the local function (if any) and the result transmitted on the remaining edge. When a message is *received* on that remaining edge, the appropriate local function products are distributed along all the other incident edges. This general procedure of gathering local function products is illustrated in Fig. 5(a).

The sum-product algorithm is made efficient by observing that the local function products gathered along paths in the factor graph can themselves be marginalized, i.e., not all variables must be *carried* along an edge. In general, a variable needs to be carried if that variable is an argument of a subsequent local function; otherwise, that variable can be marginalized out (cf. Example 3). This marginalization is carried out at the subset

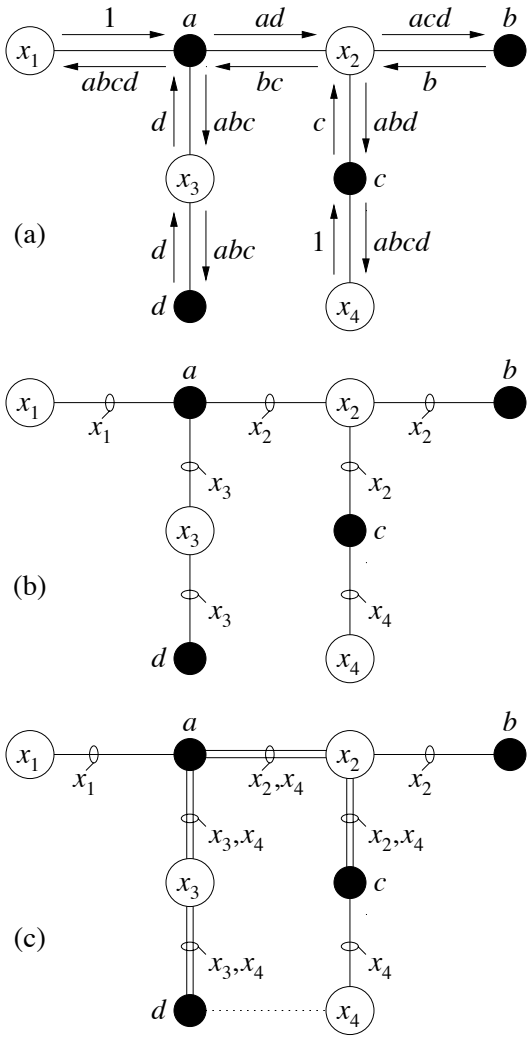

Figure 5: The sum-product algorithm: (a) local function products are "gathered" along paths from the leaves; (b) without cycles, only one variable must be "carried" along each edge; (c) with a cycle, more than one variable must be "carried" on some edges.

nodes, since these nodes are necessarily at the boundaries of the regions defining which variables are to be carried. Fig. 5(b) illustrates which variables are to be carried along each edge for the tree of Fig. 5(a).

Fig. 6 shows a fragment of a specific factor graph, which we assume forms a part of a larger tree. The update rules for this fragment are as follows:

variable to subset: (the product rule)

$$\mu_{x \to A}(x) = \mu_{B \to x}(x) \cdot \mu_{C \to x}(x) \tag{10}$$

subset to variable: (the sum-product rule)

$$\mu_{A \to x}(x) = \sum_{y,z} f_A(x, y, z) \cdot \mu_{y \to A}(y) \cdot \mu_{z \to A}(z). \tag{11}$$

termination:

$$F_x(x) = \mu_{x \to A}(x) \cdot \mu_{A \to x}(x) \tag{12}$$

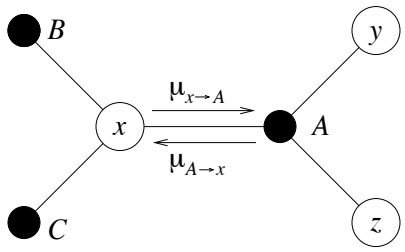

Figure 6: A factor graph fragment, showing the update rules in this case.

3.3 Dealing with Cycles

The algorithm described above terminates only if the factor graph is a tree, i.e., contains no cycles. Should the factor graph contain cycles, then these update rules will lead to the endless propagation of messages around the cycles. Although these messages may "converge" in some sense, when the factor graph has cycles, the values to which the messages converge cannot be interpreted as being exact marginal functions. (Nevertheless, in the decoding of turbo codes and low-density parity-check codes, this "iterative" algorithm *does* provide excellent performance. See Section 5.)

We now give an example of how this endless looping of messages can be dealt with so that exact marginal functions are obtained.

Example 4. Let us consider now the case where the factor graph contains a single cycle. We consider the global function

$$g(x_1, x_2, x_3) = a(x_1, x_2)b(x_2, x_3)c(x_1, x_3). \tag{13}$$

The corresponding factor graph is shown in Fig. 7(a).

To perform exact marginalization, we construct a tree spanning the factor graph. We define a variable x to be "involved" with every subset node E for which $x \in E$. In general, x will be involved with several subset nodes. We say that x must be "carried" over all edges in the spanning tree that are on a path between the nodes with which x is involved. The variable x must not be marginalized out, except when it must no longer be carried. Provided that this rule is followed, exact marginalization is possible.

Thus, if we form the spanning tree shown in Fig. 7(b) (by cutting the cycle in the factor graph at the location indicated), we see that the variable x_3 must be carried over *all* edges in the tree, but that x_1 and x_2 need only be carried over a single edge, respectively.

Fig. 5(c) illustrates this situation for the loopy factor graph that includes the dotted

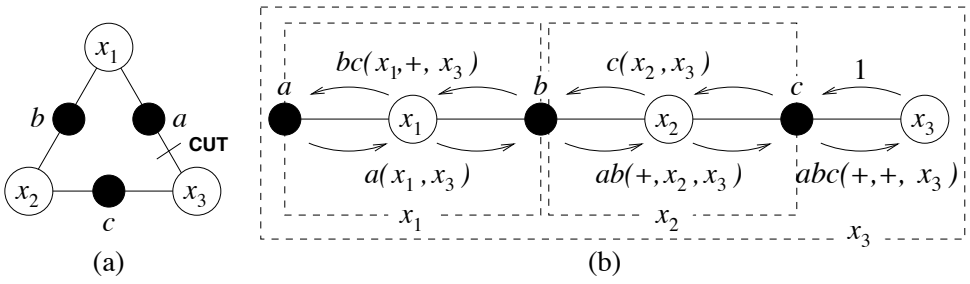

Figure 7: A spanning tree (b) for the factor graph (a), showing that x_3 must be carried everywhere.

edge. If the dotted edge is omitted to form a spanning tree, we see that the variable x_4 must be carried along edges beyond just those incident on variable node x_4. In effect, the "thickness," i.e., the number of variables to be carried, of some edges must be increased to perform an exact marginalization. In a factor graph that is a tree, each edge has unit thickness.

In general, there will be many different trees that span a given "loopy" factor graph, each of which will yield a different marginalization algorithm.

3.4 Message Passing Schedules

Thus far, we have only discussed the "two-way schedule," in which message propagate from the leaf nodes in the factor graph. In some implementations, it may be possible that message passing can occur concurrently, while in other implementations, the messages will of necessity be passed serially. In general a variety of message passing *schedules* are possible. In [12], two such schedules—the two-way schedule, and the flooding schedule—are described. The two-way schedule is best suited for a serial implementation, as it causes the smallest possible total number of messages to be sent; namely, $2E$, where E is the number of edges in the spanning tree, whereas the flooding schedule may lead to a faster convergence when the factor graph has cycles. We refer the reader to [12] for a description of these message-passing schedules.

3.5 Generalization to other Semirings

Up to now, we have assumed that all products and sums are computed in the field of real numbers. Obviously, this can be extended to an arbitrary ring. Indeed, as pointed out by Verdú and Poor [1] and other authors (notably McEliece [2, 4]; see also [3]), the appropriate algebraic structure is that of a semiring, equipped with associative and commutative '$+$' and '$\times$' operations, and a distributive law that permits distribution of '$\times$' over '$+$'.

Probably the most relevant such semiring for applications in decoding is the one that replaces '$\times$' with real addition and '$+$' with 'max' or 'min'. In channel coding for memoryless channels, if we associate with each value that a variable can take on a *cost* equal to the negative log-likelihood of the corresponding channel output, then we can associate with each valid configuration a cost equal to the sum of the costs of the variables. The minimum cost valid configuration is the "maximum-likelihood" sequence, the cost of which is computed (in a factor graph containing no cycles) by the "min-sum" generalization of the "sum-product" algorithm. The "min-sum" formulation is also the framework in which so-called "nonserial" dynamic programming is posed; see [21] for a text book treatment. See [6] for a discussion of the min-sum algorithm.

4 Examples

We now give a few examples of how the sum-product algorithm may be applied in practice.

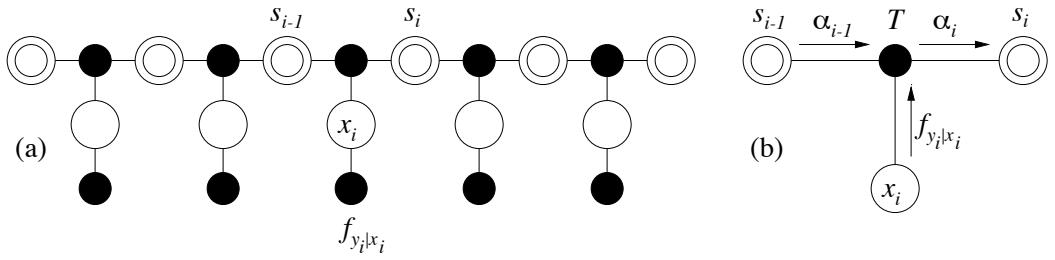

Figure 8: The factor graph for a hidden Markov model (trellis).

4.1 Hidden Markov Models and Trellis Processing

Figure 8(a) shows the factor graph representation for a hidden Markov model. State variables (not observed) are indicated with a double circle. The local function $T(s_{i-1}, x_i, s_i)$ is an indicator function that indicates which state transitions are allowed, and also which outputs (x_i) are associated with the transitions. The output variables x_i are observed at the output of a memoryless channel. Corresponding to symbol x_i, symbol y_i is observed.

Since this factor graph is a tree, exact marginalization can be achieved via the sum-product algorithm. This marginalization algorithm is be achieved in two "sweeps" through the graph, running essentially left-to-right (forward) and right-to-left (backward).

Let $\alpha_{i-1}(s_{i-1})$ denote the message sent by state variable s_{i-1} to the local function node T, as shown in Fig. 8(b). Let $f_{y_i|x_i}(x_i)$ denote the message sent by variable node x_i. Then, the sum-product algorithm will compute the message

$$\alpha_i(s_i) = \sum_{x_i} \sum_{s_{i-1}} T(s_{i-1}, x_i, s_i) \alpha_{i-1}(s_{i-1}) f_{y_i|x_i}(x_i)$$

and send this to variable node s_i. This is precisely the "forward" step in the forward/backward algorithm described in [22]. The "backward" (β) step is described in like manner. In this way, the sum-product algorithm specializes, on a trellis, to the well known forward/backward algorithm. It is easy to see that the min-sum version specializes to the Viterbi algorithm (provided we take $T(s_{i-1}, x_i, s_i) = \infty$ for an "invalid" trellis transition). Thus, we recover the well-known trellis-processing algorithms as special cases of the sum-product algorithm.

4.2 The Fast Fourier Transform

Following Aji and McEliece [4], who develop a fast Hadamard transform using a graph-based approach, we now develop the fast Fourier transform (FFT) using factor graphs. For a complex-valued data vector with components $f[n]$, $n \in \{0, 1, \ldots, N-1\}$, the N-point discrete Fourier transform $F[k]$ is defined as

$$F[k] = \sum_{n=0}^{N-1} f[n] W^{nk}$$

where $k \in \{0, 1, \ldots, N-1\}$ and $W = \exp(-j2\pi/N)$ is a complex Nth root of unity.

We consider the case $N = 8$, with $W = \exp(-j\pi/4)$, though the approach we take generalizes in an obvious way to the case where N is an arbitrary power of two. Let

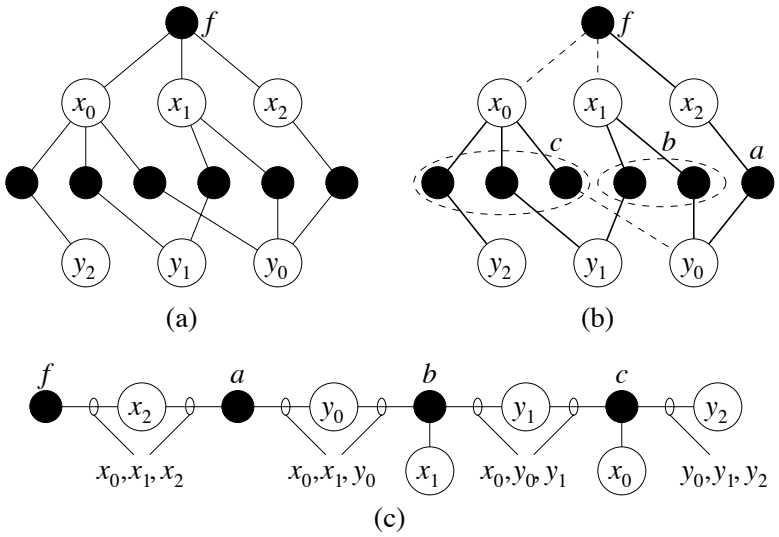

Figure 9: The discrete Fourier transform kernel: (a) factor graph; (b) a particular spanning tree; (c) spanning tree after clustering, showing which variables must be carried.

$x_0, x_1, x_2, y_0, y_1, y_2$ be binary-valued, and define $n = 4x_2 + 2x_1 + x_0$, $k = 4y_2 + 2y_1 + y_0$. We write the DFT kernel, which we take as our global function, in terms of the x_is and y_is as

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

since $W^{16} = W^8 = 1$, $W^4 = -1$, and $W^2 = -j$. The factor graph corresponding to this global function is shown in Fig. 9(a). We observe that the DFT of $f[n]$ is the marginal function $F[4y_2 + 2y_1 + y_0] = g(+, +, +, y_1, y_2, y_3)$.

The factor graph in Fig. 9(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. 9(b). (Different choices for the spanning tree will lead to possibly different DFT algorithms when the min-sum algorithm is applied.) If we cluster the local functions as shown in Fig. 9(b), essentially by defining

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

we arrive at the spanning tree shown in Fig. 9(c). Observe that three binary variables (or eight complex quantities) must be carried over each edge in the path from vertex f to vertex y_2. Along this path, 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 list of variables to be carried. In three steps, the function $f[n]$ is converted to the function $F[k]$. Clearly we have obtained a fast Fourier transform as an instance of the sum-product algorithm.

4.3 Belief Propagation in Bayesian Networks

As discussed in Section 2.4, the probability distribution (4) corresponding to a Bayesian network has the product form that allows a straightforward conversion to a factor graph

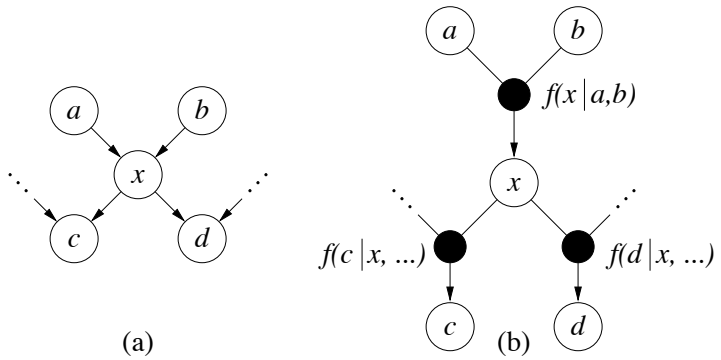

Figure 10: (a) A Bayesian network fragment with (b) the corresponding factor graph, in which, by convention, the "child" vertex in each conditional distribution is indicated with an arrow.

representation. For example, Fig. 10(a) shows a fragment of a Bayesian network, and Fig. 10(b) shows the corresponding factor graph fragment. In general, a local function, corresponding to a single factor of (4), assumes the form $f(x|\mathbf{a}(x))$, where $\mathbf{a}(x)$ is the set of "parents" of variable x in the Bayesian network. In the factor graph representation we have, by convention, identified the "child" vertex by an arrow. This allows the flow of "causality" to be inferred equally well from the factor graph as from the Bayesian network itself. Representing such causality using directed edges is often cited as one of the basic advantages of modeling a probability distribution using a Bayesian network [13]; by including arrows in the corresponding factor graph, this advantage is retained.

It is easy to show that the sum-product algorithm, when applied to cycle-free Bayesian networks, yields Pearl's belief propagation algorithm [13]. See [12] for a discussion of this correspondence.

5 Approximate "Marginalization" in Graphs with Cycles

In a great many applications, one is forced to consider factor graphs that contain cycles. In many such cases, exact marginalization in a spanning tree proves to be computationally infeasible due to the excessive "thickness" of some of the edges. Examples of factor graphs of this type include the graphs that describe low-density parity-check codes introduced by Gallager [18], and the turbo codes introduced by Berrou, *et al.* [20]. (See [3, 23] for a Tanner graph description of turbo codes, and [12,19] for a Bayesian network description.)

Nevertheless, by proceeding as if the graph contained no cycles (i.e., by applying the "carry-one-variable" sum-product algorithm), excellent results may be obtained. For example, simulation results of MacKay and Neal [17] indicate that low-density parity-check codes can be decoded (with reasonable complexity) using this approach so that the E_b/N_0 required to achieve a bit error rate of 10^{-5} is within approximately 2dB of the Shannon limit [17] (using antipodal signaling on a white Gaussian noise channel). Even more impressive are the well-known results of Berrou, *et al.*, who show that an iterative sum-product decoding procedure for turbo codes can approach the Shannon limit to within a 0.5 dB in E_b/N_0 required to achieve a bit error rate of 10^{-5}.

While a theoretical understanding of the convergence of the sum-product algorithm in graphs with cycles has not, to the best of our knowledge, been achieved in general, the excellent decoding performance obtained is undeniable. In our view, this is a great motivation, not only to achieve a theoretical understanding of the properties of the sum-product algorithm on graphs with cycles, but also to find further families of codes for which the sum-product algorithm will prove effective.

6 Conclusions

Factor graphs provide a natural description of the factorization of a global function into a product of local functions. As such, factor graphs are relevant to a broad spectrum of application areas. We have illustrated that the sum-product algorithm encompasses a wide variety of previously known algorithms, including the Viterbi algorithm, the forward/backward algorithm, Pearl's belief propagation algorithm, even the fast Fourier transform. No doubt many other algorithms can be captured in this framework. Because of its generality and essential simplicity, we feel that the sum-product algorithm should be included as a part of every engineer's standard "algorithms toolkit."

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, and R. M. Tanner. We benefitted greatly from the many discussions on this topic that took place in Ulm.

References

[1] 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.

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

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

[4] 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.

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

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

[7] G. D. Forney, Jr., "The forward-backward algorithm," in *Proc. 34th Annual Allerton Conf. on Communication, Control, and Computing*, (Allerton House, Monticello, Illinois), pp. 432–446, October 1996.

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

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

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

[11] 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.

[12] 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, Jan. 1998.

[13] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufmann, 1988. Revised second printing.

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

[15] R. E. Neapolitan, *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. Toronto: John Wiley & Sons, 1990.

[16] B. J. Frey, *Bayesian Networks for Pattern Classification, Data Compression and Channel Coding*. Toronto, Canada: Department of Electrical and Computer Engineering, University of Toronto, 1997. Doctoral dissertation available at `http://www.cs.utoronto.ca/~frey`.

[17] 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.

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

[19] 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, Jan. 1998.

[20] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. Commun. (ICC)*, (Geneva, Switzerland), pp. 1064–1070, 1993.

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

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

[23] N. Wiberg, H.-A. Loeliger, and R. Kötter, "Codes and iterative decoding on general graphs," *European Transactions on Telecommunications*, vol. 6, pp. 513–525, 1995.