Factor Graphs and the Sum-Product Algorithm

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

July 27, 1998

Abstract—A factor graph is a bipartite graph that expresses how a "global" function of many variables factors into a product of "local" functions. Factor graphs subsume many other graphical models including Bayesian networks, Markov random fields, and Tanner graphs. Following one simple computational rule, the sum-product algorithm operates in factor graphs to compute—either exactly or approximately—various marginal functions by distributed message-passing in the graph. 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. This paper is available on the web at `http://www.comm.utoronto.ca/frank/factor/`.

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

†The Beckman Institute, 405 North Mathews Avenue, Urbana, IL 61801, USA (`frey@cs.utoronto.ca`)

‡Endora Tech AG, Gartenstrasse 120, CH-4052 Basel, SWITZERLAND (`haloeliger@access.ch`)

Typeset with LaTeX at 16:16 on July 27, 1998.

Contents

1 Introduction

1.1 Factor Graphs

A *factor graph* is a bipartite graph that expresses how a "global" function of many variables factors into a product of "local" functions. Suppose, e.g., that some real-valued function $g(x_1, x_2, x_3, x_4, x_5)$ of five variables can be written as the product

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \tag{1}$$

of five functions, $f_A, f_B, \ldots, f_E$. The corresponding factor graph is shown in Fig. 1(a). There is a *variable node* for each variable, there is a *function node* for each factor, and the variable node for x_i is connected to the function node for f if and only if x_i is an argument of f.

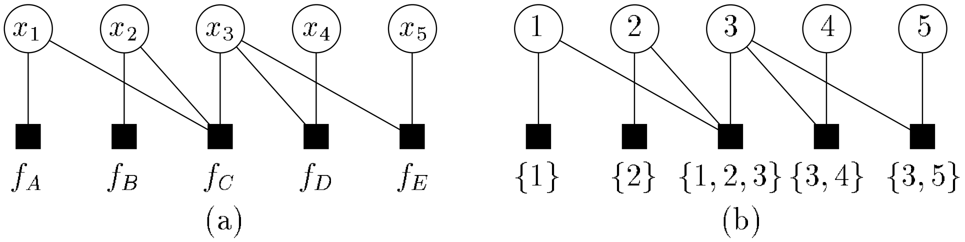


Figure 1: A factor graph that expresses that a global function factors as the product of local functions $f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5)$. (a) Variable/function view, (b) index/subset view. Variable nodes are shown as circles; function nodes are shown as filled squares.

We will use the following notation. Let $X_S = \{x_i : i \in S\}$ be a collection of variables indexed by a finite set S, where S is linearly ordered by $\leq$. For each $i \in S$, the variable x_i takes on values from some set A_i. Most often S will be a subset of the integers with the usual ordering. If E is a subset of S, then we denote by $X_E = \{x_i : i \in E\}$ to be the subset of variables indexed by E.

A particular assignment of a value to each of the variables of X_S will be referred to as a *configuration* of the variables. Configurations of the variables can be viewed as being elements of the Cartesian product $A_S \triangleq \prod_{i \in S} A_i$, called the *configuration space.* For concreteness, we suppose that the components of configurations are ordered as in S, so that if $S = \{i_1, i_2, \ldots, i_N\}$ with $i_1 \leq i_2 \leq \cdots \leq i_N$, then a typical configuration a is written as $a = (a_{i_1}, a_{i_2}, \ldots, a_{i_N})$ with $a_{i_j} \in A_{i_j}$ for $j = 1, \ldots, N$. Of course, the configuration a is equivalent to the multiple variable assignment $x_{i_1} = a_{i_1}$, $x_{i_2} = a_{i_2}$, $\ldots$, and *vice versa.* By abuse of notation, if a is a particular configuration, we will write $X_S = a$ for this assignment. We will have occasion to view configurations both as assignments of values to variables and as elements of A_S.

We will also have occasion to consider *subconfigurations*: if $E = \{j_1, j_2, \ldots, j_M\} \subset S$, with $j_1 \leq j_2 \leq \cdots \leq j_M$, and a is any configuration, then the M-tuple $a_E = (a_{j_1}, \ldots, a_{j_M})$ is called the subconfiguration of a with respect to E. The set $\{a_E : a \in A_S\}$ of all subconfigurations with respect to E is denoted by A_E; clearly $A_E = \prod_{i \in E} A_i$. Again, by abuse of notation, if a_E is a particular subconfiguration with respect to E, we will write $X_E = a_E$ for the multiple variable assignment $x_{j_1} = a_{j_1}$, $x_{j_2} = a_{j_2}$, etc.

Finally, if $C \subset A_S$ is some set of configurations, we will denote by C_E the set of subconfigurations of the elements of C with respect to E, i.e., $C_E = \{a_E : a \in C\}$. Clearly $C_E \subset A_E$.

Let $g\colon A_S \to R$ be a function with the elements of X_S as arguments. For the moment, we require the domain R of g to be equipped with a binary product (denoted '$\cdot$') and a unit element (denoted 1) satisfying, for all u, v, and w in R,

$$1 \cdot u = u, \qquad u \cdot v = v \cdot u, \qquad (u \cdot v) \cdot w = u \cdot (v \cdot w),$$

so that R forms a commutative semigroup with unity. The reader will lose nothing essential in most cases by assuming that R is a field, e.g., the real numbers, under the usual product. We will usually denote the product of elements x and y by the juxtaposition xy, and only occasionally as $x \cdot y$.

Suppose, for some collection Q of subsets of S, that the function g factors as

$$g(X_S) = \prod_{E \in Q} f_E(X_E) \tag{2}$$

where, for each $E \in Q$, $f_E\colon A_E \to R$ is a function of the subconfigurations with respect to E. We refer to each factor $f_E(X_E)$ in (2) as a *local function.* (If some $E \in Q$ is empty, i.e., $E = \emptyset$, we interpret the corresponding local "function" $f_\emptyset$ with no arguments as a constant in R.) Often, as is common practice in probability theory, we will use an abbreviated notation in which the arguments of a function determine the function domain, so that, e.g., $f(x_1, x_2)$ would denote a function from $A_1 \times A_2 \to R$, as would $f(x_2, x_1)$. In this abbreviated notation, (2) would be written as $g(X_S) = \prod_{E \in Q} f(X_E)$.

A factor graph representation of (2) is a bipartite graph denoted $F(S, Q)$ with vertex set $S \cup Q$ and edge set $\{\{i, E\} : i \in S, E \in Q, i \in E\}$. In words, $F(S, Q)$ contains an edge $\{i, E\}$ if and only if $i \in E$, i.e., if and only if x_i is an argument of the local function f_E. Those vertices that are elements of S are called *variable nodes* and those vertices that are elements of Q are called *function nodes.* For example, in (1), we have $S = \{1, 2, 3, 4, 5\}$ and $Q = \{\{1\}, \{2\}, \{1, 2, 3\}, \{3, 4\}, \{3, 5\}\}$, which gives the factor graph $F(S, Q)$ shown in Fig. 1(b). Throughout, we will translate freely between factor graphs labeled with variables and local functions (the 'variable/local function view' of Fig. 1(a)) and the corresponding factor graph labeled with variable indices and index subsets (the 'index/subset view' of Fig. 1(b)). To avoid the more precise but often quite tedious mention of functions "corresponding to" function nodes, and variables "corresponding to" variable nodes, we will blur the distinction between the nodes and the objects associated with them, thereby making it legitimate to refer to, say, the arguments of a function node f, or the edges incident on a variable x_i.

It will often be useful to refer to an arbitrary edge of a factor graph. Such an edge $\{v, w\}$ by definition is incident on a function node and a variable node; the latter is called the variable *associated* with the given edge, and is denoted by $x_{\{v,w\}}$.

1.2 Prior Art

We will see in Section 2 that factor graphs subsume many other graphical models in signal processing, probability theory, and coding, including Markov random fields [19, 21, 32], Bayesian networks [20, 31] and Tanner graphs [35, 38, 39]. Our original motivation for introducing factor graphs was to make explicit the commonalities between Bayesian networks (also known as belief networks, causal networks, and influence diagrams) and Tanner graphs, both of which had previously been used to explain the iterative decoding of turbo codes and low-density parity check codes [11, 22, 25, 26, 30, 38, 39]. In that respect, factor graphs and their applications to coding are just a slight reformulation of the approach of Wiberg, *et al.* [39]. However, a main thesis of this paper is that factor graphs may naturally be used in a wide variety of fields other than coding, including signal processing, system theory, expert systems, and artificial neural networks.

It is plausible that many algorithms in these fields are naturally expressed in terms of factor graphs. In this paper we will consider only one such algorithm: the *sum-product algorithm*, which operates in a factor graph by passing "messages" along the edges of the graph, following a single, simple, computational rule. (By way of preview, a very simple example of the operation of the sum-product algorithm operating in the factor graph of Fig. 1 is given in the next subsection.)

The main purpose of this essentially tutorial paper is to illuminate the simplicity of the sum-product algorithm in the general factor graph setting, and then point out a variety of applications. In parallel with the development of this paper, Aji and McEliece [1, 2] develop the closely related "generalized distributive law," an alternative approach based on the properties of junction trees (and not factor graphs). Aji and McEliece also point out the commonalities among a wide variety of algorithms, and furnish an extensive bibliography. Forney [10] gives a nice overview of the development of many of these algorithms, with an emphasis on applications in coding theory.

The first appearance of the sum-product algorithm in the coding theory literature is probably Gallager's decoding algorithm for low-density parity-check codes [14]. The optimum (minimum probability of symbol error) detection algorithm for codes, sometimes referred to as the MAP (maximum *a posteriori* probability) algorithm or the BCJR algorithm (after the authors of [4]) turns out to be a special case of the sum-product algorithm applied to a trellis. This algorithm was developed earlier in the statistics literature [5] and perhaps even earlier in classified work due to L. R. Welch [29]. In the signal processing literature, and particularly in speech processing, this algorithm is widely known as the forward-backward algorithm [33]. Pearl's belief propagation and belief revision algorithms, widely applied in expert systems and in artificial intelligence, turn out to be examples of the sum-product algorithm operating in a Bayesian network; see [31, 20] for textbook treatments. Neural network formulations of factor graphs have also been used for unsupervised learning and density estimation; see, e.g., [11].

In coding theory, Tanner [35] generalized Gallager's bipartite graph approach to low-complexity codes and also developed versions of the sum-product algorithm. Tanner's approach was later generalized to graphs with hidden (state) variables by Wiberg, *et al.* [39, 38]. In coding theory, much of the current interest in so-called "soft-output" decoding algorithms stems from the near-capacity-achieving performance of turbo codes, introduced by Berrou, *et al.* [7]. The turbo decoding algorithm was formulated in a Bayesian network framework by McEliece, *et al.* [30], and Kschischang and Frey [22].

1.3 A Sum-Product Algorithm Example

As we will describe more precisely in Section 4, the sum-product algorithm can be used (in a factor graph that forms a tree) to compute a function *summary* or *marginal.* For example, consider the specific case in which the global function defined in (1) is real-valued and represents the conditional joint probability mass function of a collection of discrete random variables, given some observation y. We might be interested in, say, the marginal function

$$p(x_1|y) = \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} g(x_1, x_2, x_3, x_4, x_5).$$

From the factorization given by (1), we write

$$\begin{aligned}
p(x_1|y) &= \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \\
&= f_A(x_1) \underbrace{\sum_{x_2} f_B(x_2) \sum_{x_3} f_C(x_1, x_2, x_3) \underbrace{\underbrace{\sum_{x_4} f_D(x_3, x_4)}_{f_D(x_3,x_4)\downarrow x_3} \underbrace{\sum_{x_5} f_E(x_3, x_5)}_{f_E(x_3,x_5)\downarrow x_3}}_{f_{DE}(x_3,x_4,x_5)\downarrow x_3}}_{f_{BCDE}(x_1,x_2,x_3,x_4,x_5)\downarrow x_1}
\end{aligned} \tag{3}$$

where we write $f_{DE}(x_3, x_4, x_5)$ for the product $f_D(x_3, x_4) f_E(x_3, x_5)$, and $f_{BCDE}(x_1, x_2, x_3, x_4, x_5)$ for the product $f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5)$.

In (3) we have identified the various factors that need to be computed to obtain $p(x_1|y)$. We have used a notation for a *summary operator* that will be introduced in Section 3. In this example, for $i \in E$, the notation $f(X_E) \downarrow x_i$, called the summary of $f(X_E)$ for x_i, is defined as the marginal function

$$f(X_E) \downarrow x_i = \sum_{x_j : j \in E \setminus \{i\}} f(X_E)$$

obtained by summing over all possible subconfigurations of the arguments of f, *other* than x_i. In this notation, $p(x_1|y) = g(X_S) \downarrow x_1$.

Our primary observation is that $g(X_S) \downarrow x_1$ can be computed knowing just $f_A(x_1)$ and $f_{BCDE}(x_1) \downarrow x_1$. The latter factor can be computed knowing just $f_B(x_2)$, $f_C(x_1, x_2, x_3)$ and $f_{DE}(x_3, x_4, x_5) \downarrow x_3$. In turn, $f_{DE}(x_3, x_4, x_5) \downarrow x_3$ can be computed knowing just $f_D(x_3, x_4) \downarrow x_3$ and $f_E(x_3, x_4) \downarrow x_4$.

These products can be "gathered" in a distributed manner in the factor graph for g, as shown in Fig. 2. Imagine a processor associated with each node of the factor graph, capable of performing local computations (i.e., computing local function products and local function summaries), and imagine also that these processors are capable of communicating with adjacent processors by sending "messages" along the edges of the factor graph. The messages are descriptions of summaries of various local function products. By passing messages as shown in Fig. 2, all of the factors necessary for the computation of $g(X_S) \downarrow x_1$ become available at x_1.

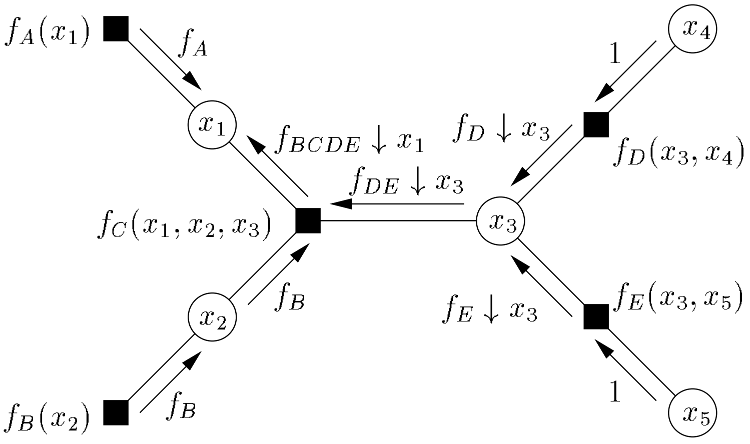


Figure 2: Computing $g(x_1, \ldots, x_5) \downarrow x_1$ by the sum-product algorithm.

As we will make clear later, each processor needs only to follow a single, simple, computational rule: the message passed from node v to node w on the edge $\{v, w\}$ is the product of the messages that arrive on all *other* edges incident on v with the local function at v (if any), summarized for the variable $x_{\{v,w\}}$ associated with the given edge. The reader may verify that precisely this rule was followed in generating the messages passed in the factor graph of Fig. 2. In a tree, the algorithm terminates by computing at a variable node (x_1, in this case) the product of all incoming messages.

As an exercise, the reader may wish to verify that the various factors needed to compute *each* marginal function can be obtained as products of the messages sent by the sum-product algorithm, as is shown in Fig. 18 of Section 4. From this conceptually simple computational procedure operating in the corresponding factor graph, we will be able to derive the wide variety of algorithms mentioned in the Abstract.

1.4 Notational Preliminaries

We will need the following terminology and ideas from graph theory. Let $G(V, E)$ be a graph with vertex set V and edge set E. Edges are not directed; an edge between two vertices v and w is the unordered pair $e = \{v, w\}$, and e is said to be *incident on* v and w. The *degree* $\partial(v)$ of a vertex v is the number of edges incident on v. For every v, the set $n(v)$ of *neighbors* of v is $n(v) = \{w : \{v, w\} \in E\}$, the set of vertices that share an edge with v. Clearly v has $\partial(v)$ distinct neighbors. If f is a function node in a factor graph, then $X_{n(f)}$ is the set of arguments of f.

A *path* between vertices v and w in G is defined, as usual, as a sequence of distinct edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{L-1}, v_L\}$ in E with $v_1 = v$, and $v_L = w$. A *cycle* in G is a path between a vertex v and itself. If there is a path between v and w, then v and w are said to be *connected* in G. Since vertices are always considered to be connected to themselves, connectedness is an equivalence relation on the vertices of G; the disjoint equivalence classes induce disjoint subgraphs of G called the (connected) *components* of G. If G comprises a single component—as will be the case for the majority of factor graphs considered in this paper—then G is said to be *connected.*

A graph G is a *tree* if it is connected and has no cycles. A graph G is a tree if and only if there is a unique path between any pair of distinct vertices in G. In a tree, a vertex v is said to be a *leaf node* if $\partial(v) \leq 1$. In any finite tree of more than one node, there are always at least two leaf nodes. If u and w are two arbitrary but distinct vertices in a tree G, and v is a leaf node distinct from u and w, then the path from u to w does not pass through v. Since u and w are arbitrary (though distinct from v) this means that if v and the edge incident on v are deleted from G, the resulting graph is still a tree.

More generally, if G is a tree then the graph obtained by cutting (i.e., removing) any edge $\{v, w\}$ from G is the union of two components, one denoted $G_{w \to v}$ (containing v, but not w) and the other denoted $G_{v \to w}$ (containing w but not v), both of which are themselves trees. The notation is intended to be mnemonic: $G_{w \to v}$ is the subgraph of G as "viewed" from the edge $\{v, w\}$ while facing in the direction of v.

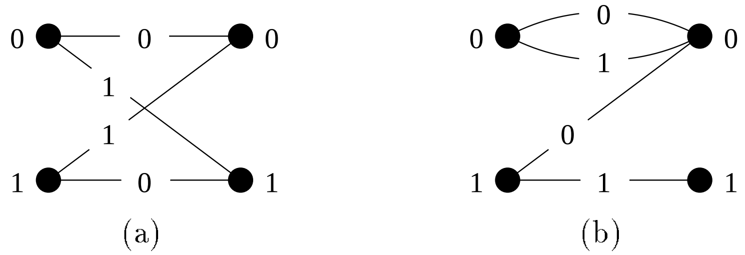
Variables corresponding to nodes in distinct components of a factor graph F are said to *contribute independently* to the corresponding (global) function. More generally, two variable subsets X_E and $X_{E'}$ contribute independently to g if E and E' are contained in distinct components of F. If g is the joint probability mass or density function of a collection of random variables, and if x_i and x_j are variables that contribute independently to g, then the corresponding random variables are independent. (The converse is not necessarily true.)

We will also need the following useful notation called "Iverson's convention" [15, p. 24] for indicating the truth of a logical proposition: if P is a Boolean proposition, then $[P]$ is the binary function that indicates whether or not P is true, i.e.,

$$[P] = \begin{cases} 1 & \text{if } P; \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

We will use Iverson's convention in formulas only in those contexts in which it is sensible to have a $\{0,1\}$-valued quantity. We will occasionally use square brackets simply as brackets, but this should cause no confusion since the enclosed quantity will in those cases not be a Boolean proposition.

We will often have occasion to consider binary indicator functions, i.e., $\{0,1\}$-valued functions of several variables. A convenient graphical representation for a binary indicator function of three variables taking values in finite alphabets is a *trellis section.* If $f(x,y,z)$ is such a function, then there exists a set $B \subset A_x \times A_y \times A_z$ such that $f(x,y,z) = [(x,y,z) \in B]$. We take the set B as defining the set of labeled edges in a directed bipartite graph, called a trellis section. We take the set A_x as the set of "left vertices," the set A_z as the set of "right vertices," and the set A_y as the set of "edge labels." Each triple $(x,y,z) \in B$ defines an edge with left vertex x, right vertex z, and label y. For example, for binary variables x, y, and z, (considered as elements of the field $GF(2)$) the trellis sections corresponding to $[x+y=z]$ and $[xy=z]$ are shown in Fig. 3(a) and (b), respectively. In this figure, as in all of our figures of trellis sections, we have placed the left vertices on the left and the right vertices on the right, so that an arrow indicating the orientation of an edge is not required.



(a)

(b)

Figure 3: Trellis sections corresponding to binary indicator functions of three variables: (a) $[x+y=z]$, (b) $[xy=z]$.

1.5 Organization of the Paper

The remainder of this paper is organized as follows. In Section 2, to illustrate the broad applicability of factor graphs, and to motivate the remainder of the paper, we provide a number of examples of factor graphs in a variety of areas including coding theory, systems theory, probability theory, neural networks and signal processing.

In Section 3, we study cross-sections and projections of functions, and formulate the general notion of a "summary operator" that acts on a function projection to create marginal functions.

The operation of the sum-product algorithm is described in Section 4. As we have already briefly described, the sum-product algorithm operates using a local computational procedure that is characterized by one conceptually simple computational rule. In a tree, we prove that this procedure results in exact function marginalization. (In Appendix B we provide a complexity analysis for the important case in which the summary operator is defined in terms of a sum operation like that in the example of Section 1.3.)

In Section 5 we apply the sum-product algorithm to the factor graphs of Section 2, and obtain a variety of well-known algorithms as special cases. These include the forward/backward algorithm, the Viterbi algorithm, Pearl's belief propagation algorithms for Bayesian networks, and the Kalman filter.

In Section 6, we describe some of the possible transformations that may be applied to a factor graph without changing the function that it represents. This will be used to motivate a procedure for exact marginalization in factor graphs with cycles. As an application of this procedure, we derive a Fast Fourier Transform algorithm as a special case of the sum-product algorithm.

Some concluding remarks are offered in Section 7.

2 Examples of Factor Graphs

Having defined the general concept of factor graphs, we now give some examples of the way in which factor graphs may be used to represent useful functions. In Section 5, we will describe some of the applications of the sum-product algorithm using these graphs.

Among all multi-variable functions that we might wish to represent by a factor graph, two particular classes stand out: set membership *indicator functions,* whose value is either 0 or 1, and *probability distributions.* Such functions are often interpreted as models—set theoretic or probabilistic, respectively—of a physical system. For example, Willems' system theory [40] starts from the view that a "system" (i.e., a model) *is* simply a set of allowed trajectories in some configuration space. Factorizations of such functions can give important structural information about the model. Moreover, the structure of the factor graph has a strong influence on the performance of the sum-product algorithm; e.g., we will see in Section 4 that the algorithm is "exact," in a well-defined sense, only if the graph has no cycles. We shall see below that a number of established modeling styles, both set theoretic and probabilistic, correspond to factor graphs with a particular structure.

2.1 Indicating Set Membership: Tanner Graphs

We start with set membership indicator functions. As usual let S be an index set, and let A_S denote a configuration space. In many applications, particularly in coding theory, we work with a fixed subset C of A_S, which we think of as the set of codewords or *valid configurations*. In these applications, we will be interested in the set membership indicator function

$$g(X_S)\colon A_S \to \{0,1\}$$

defined, for all $a \in A_S$, by

$$g(a) = [a \in C] = \begin{cases} 1 & \text{if } a \in C; \\ 0 & \text{otherwise.} \end{cases}$$

Of course, in this paper, we will be interested in situations in which $g(X_S)$ factors as in (2). In particular, suppose that each factor, $f_E(X_E)$, is itself a binary indicator function, that indicates whether a particular subconfiguration a_E is "locally valid." Both the global function and all local functions take values in the set $\{0,1\}$, considered as a subset of the reals under the usual multiplication. In this setup, a configuration $a \in A_S$ is valid (i.e., $g(a) = 1$) if and only if $f_E(a_E) = 1$ for all $E \in Q$. The product acts as a logical conjunction (AND) operator: a (global) configuration is valid if and only if all of its subconfigurations are valid.

In this context, local functions are often referred to as (local) *checks*, and the corresponding function nodes in the factor graph are also called *check nodes*. Given a code $C \subset A_S$, we will often (somewhat loosely) refer to a factor graph representation for C; what we strictly mean in such situations is a factor graph representation for the indicator function $g(a) = [a \in C]$. We will often refer to a factor graph for a set membership indicator function that factors as a product of local checks as a *Tanner graph*.

Example 1. (*Linear codes*)
Of course, every code has a factor graph representation (and in general more than one). A

convenient way to construct a factor graph for a *linear* code is to start with a parity-check matrix for the code.

To illustrate, consider the linear code C over $GF(2)$, defined by the parity check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}, \tag{5}$$

consisting of of all binary 6-tuples $\mathbf{x} \triangleq (x_1, x_2, \ldots, x_6)$ satisfying $H\mathbf{x}^T = 0$. Since every linear code has a parity-check matrix, this approach applies to all linear codes.

In effect, each row of the parity-check matrix gives us an equation that must be satisfied by $\mathbf{x}$, and $\mathbf{x} \in C$ if and only if *all* equations are satisfied. Thus, if a binary function indicating satisfaction of each equation is introduced, the product of these functions indicates membership in the code.

In our example, H has three rows, and hence the code membership indicator function $\mu(x_1, x_2, \ldots, x_6)$ can be written as a product of three local indicator functions:

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

where we have again used Iverson's convention and where $\oplus$ denotes the sum in $GF(2)$. The corresponding factor graph is shown in Fig. 4.

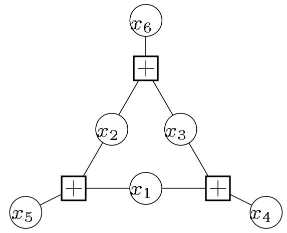


Figure 4: A factor graph for the binary linear code of Example 1.

In Fig. 4, we have used a special symbol for the parity checks (a square with a "+" sign instead of a black square). In fact, we will freely use a variety of symbols for function nodes, depending on the type of local function. Variable nodes will always be drawn as circles; double circles (as in Fig. 6, described below) will sometimes be used to indicate auxiliary variables (states).

Example 2. (*Logic circuits*)
Many readers may be surprised to note that they are already quite familiar with certain

factor graphs, for example, the factor graph shown in Fig. 5. Here, the local checks are drawn as logic gates, to remind us of the definition of the corresponding binary indicator function. For example, the AND gate with inputs u_1 and u_2 and output x_1 represents the binary indicator function $f(u_1, u_2, x_1) = [x_1 = u_1 \text{ AND } u_2]$.

Viewed as a factor graph, the logic circuit of Fig. 5 represents the global function

$$g(u_1, u_2, u_3, u_4, x_1, x_2, y) = [x_1 = u_1 \text{ AND } u_2][x_2 = u_3 \text{ AND } u_4][y = x_1 \text{ OR } x_2]. \quad (6)$$

The function g takes on the value 1 if and only if its arguments form a configuration consistent with the correct functioning of the logic circuit.

In general, any logic circuit can be viewed as a factor graph. The local function corresponding to some elementary circuit takes on the value 1 if and only if the corresponding variables behave (locally) as required by the circuit. If necessary, auxiliary variables (not directly observable as input or outputs) like x_1 and x_2 in Fig. 5 may be introduced between logic gates. As we shall see in the next example, the introduction of such auxiliary variables can give rise to particularly "nice" representations of set-membership indicator functions.

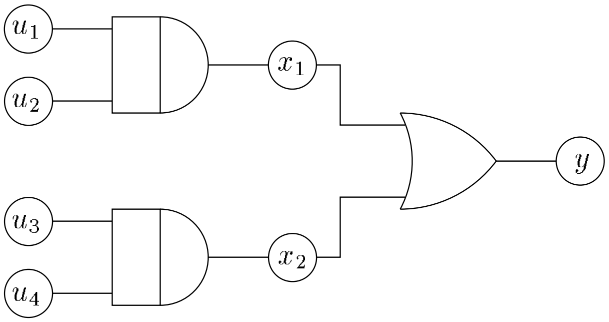


Figure 5: A logic circuit—also a factor graph!

Example 3. (*Auxiliary variables*)
When dealing with binary functions indicating membership in some set $C \subset A_S$ of valid configurations, it will often be useful to introduce *auxiliary* or *state* or *hidden* variables. By this, we mean the introduction of a set $T \supset S$, and a set $X_{T \setminus S}$ of variables indexed by T but not by S, called auxiliary variables.

In this setup, we view A_S as being the set of subconfigurations with respect to S of the enlarged configuration space A_T. We can then introduce a set $D \subset A_T$ of configurations in the enlarged space. Provided that $D_S = C$, i.e., that the subconfigurations of the elements of D with respect to S is equal to C, we will consider a factor graph for D to be a valid factor graph for C. Following Forney [10] we will sometimes refer to such a factor graph—and any factor graph for a set membership indicator function having auxiliary variables—as as a TWL (Tanner/Wiberg/Loeliger) graph for C. As mentioned earlier, auxiliary variable nodes are indicated with a double circle in our factor graph diagrams.

To illustrate this idea, Fig. 6(b) shows a TWL graph for the binary code of Example 1. In addition to the variable nodes x_1, x_2, ... , x_6, there are also variable nodes for the auxiliary variables s_0, s_1, ... , s_6. The definition of the local checks, which are drawn as generic function nodes (black squares), is given in terms of a trellis for the code, which is shown in Fig. 6(a).

A trellis for C is defined by the property that the sequence of edge labels encountered in each directed path (left to right) from the leftmost vertex to the rightmost vertex in the trellis is always a codeword in C, and that each codeword is represented by at least one such path.

Here, the auxiliary variables s_0, ... , s_6 correspond to the trellis states, and each local check represents one trellis section, i.e., the ith local function (counting from the left) indicates which triples (s_{i-1}, x_i, s_i) are valid (state, output, next state) transitions in the trellis, drawn according to our convention for indicator functions of three variables introduced in Section 1.

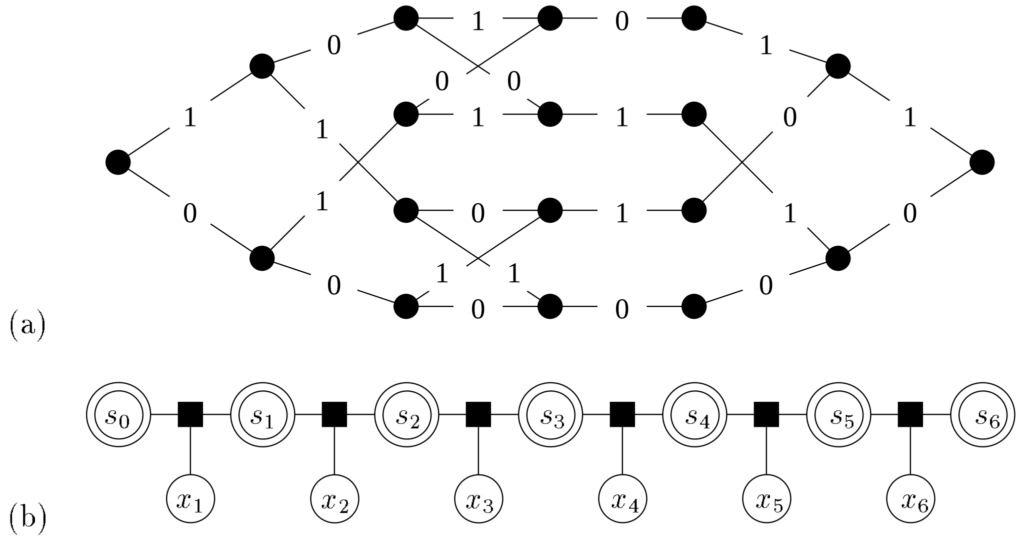


Figure 6: A trellis (a) and the corresponding TWL graph (b) for the code of Fig. 4.

In this example, the second trellis section from the left in Fig. 6 consists of the following triples (s_1, x_2, s_2):

$$S_2 = \{(0,0,0), (0,1,2), (1,1,1), (1,0,3)\}, \tag{7}$$

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. 6(a). The corresponding check (i.e., local function) in the TWL graph is the indicator function $f(s_1, x_2, s_2) = [(s_1, x_2, s_2) \in S_2]$.

The method described in this example to obtain a factor graph from a trellis is completely general and applies to any trellis. Since every code can be represented by a trellis (see [36] for a recent survey of results in the theory of the trellis structure of codes), this shows that a cycle-free factor graph exists for every code (in fact, for every set membership function).

In general, the purpose of introducing auxiliary variables (states) is to obtain "nice" factorizations of the global function that are not otherwise possible.

Example 4. (*State-space models*)
Trellises are convenient representations for a variety of signal models. For example, the generic factor graph of Fig. 7 can represent any time-invariant (or indeed, time-varying) state space model. As in Fig. 6, each local check represents a trellis section, an indicator function for the set of allowed combinations of left state, input symbol, output symbol, and right state. (Here, a trellis edge has two labels.)

For example, the classical linear state space model is given by the equations

$$\begin{array}{rcrcl} x(j+1) & = & Ax(j) & + & Bu(j), \\ y(j) & = & Cx(j) & + & Du(j), \end{array} \tag{8}$$

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. 7. 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 (8).

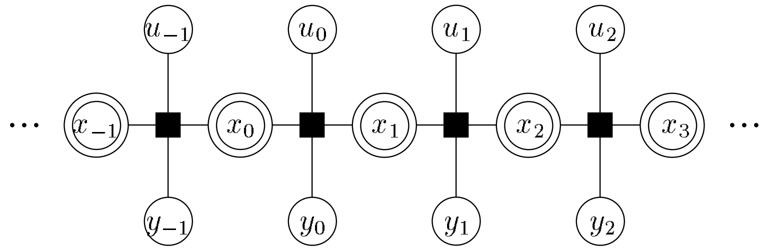


Figure 7: Generic factor graph for a time-invariant trellis.

2.2 Probability Distributions

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. (*Decoding*)
Consider the situation most often modeled in coding theory, in which a codeword $(x_1, \ldots, x_n)$ is selected with uniform probability from a code C of length n, and transmitted over a discrete memoryless channel with corresponding output $(y_1, \ldots, y_n)$. Since the channel is memoryless, by definition the conditional probability mass or density function evaluated at a particular channel output assumes the product form:

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

The *a priori* probability of selecting a particular codeword is a constant, and hence the *a priori* joint probability mass function for the codeword symbols is proportional to the code set membership indicator function. It follows that the joint probability mass function of $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ is proportional to

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = [(x_1, \ldots, x_n) \in C] \prod_{i=1}^{n} f(y_i | x_i). \qquad (9)$$

Of course, as described in the previous subsection, the code membership indicator function itself may factor into a product of local indicator functions. For example if C is the binary linear code of Example 1, we have

$$\begin{aligned} f(x_1, \ldots, x_6, y_1, \ldots, y_6) \;=\; & [x_1 \oplus x_2 \oplus x_5 = 0] \cdot [x_2 \oplus x_3 \oplus x_6 = 0] \cdot [x_1 \oplus x_3 \oplus x_4 = 0] \cdot \\ & \prod_{i=1}^{6} f(y_i | x_i), \end{aligned}$$

whose factor graph is shown in Fig. 8(a). We see that a factor graph for the joint probability mass function of codeword symbols and channel output symbols is obtained simply by augmenting the factor graph for the code itself.

In decoding, we invariably work with the conditional joint probability mass function for the codeword symbols given the observation of the channel output symbols. As will be discussed in more detail in Section 3.1, in general function terms we deal with a (scaled)

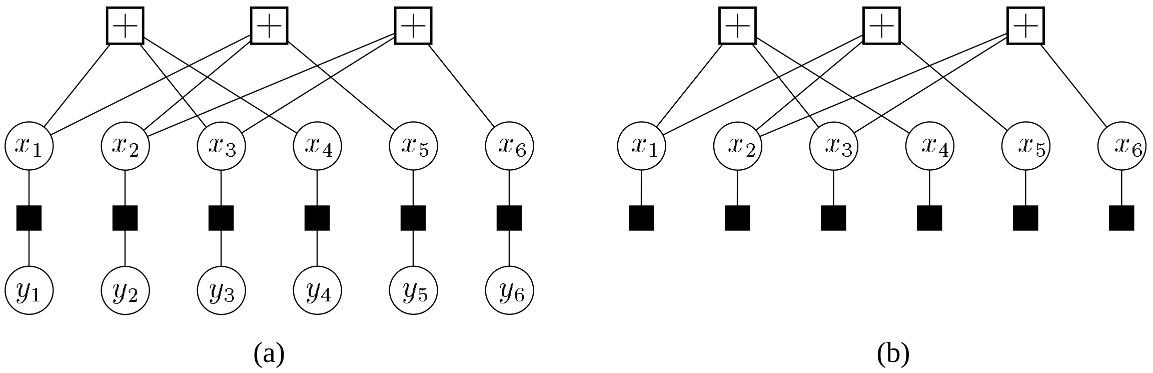


Figure 8: Factor graphs for (a) the joint probability density function of channel input and output for the binary linear code of Fig. 4, (b) the cross-section after observation of a particular channel output vector.

cross-section of the joint probability mass function defined in (9). It turns out that a factor graph for a function cross-section is obtained from a factor graph for the function simply by removing the nodes corresponding to the observed variables, and replacing all local functions with their cross-sections; this is shown for our example code in Fig. 8(b). The cross-section of the function $f(y_i|x_i)$ can be interpreted as a function of x_i with *parameter* y_i.

Example 6. (*Markov chains, hidden Markov models, and factor graphs with arrows*) In general, let $f(x_1, \ldots, x_n)$ denote the joint probability mass function of a collection of random variables. By the chain rule of conditional probability, we may always express this function as

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

For example if $n = 4$, 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. 9(b).

Because of this chain rule factorization, in situations involving probability distributions, we will often have local functions of the form $f(x_i|\mathbf{a}(x_i))$, where $\mathbf{a}(x_i)$ is some collection of variables referred to as the *parents* of x_i. In this case, motivated by a similar convention in Bayesian networks [20, 31] (see example 8, below), we will sometimes indicate the *child*, i.e., x_i, in this relationship by placing an arrow on the edge leading from the local function f to x_i. We have followed this *arrow convention* in Fig. 9(b)–(d), and elsewhere in this paper.

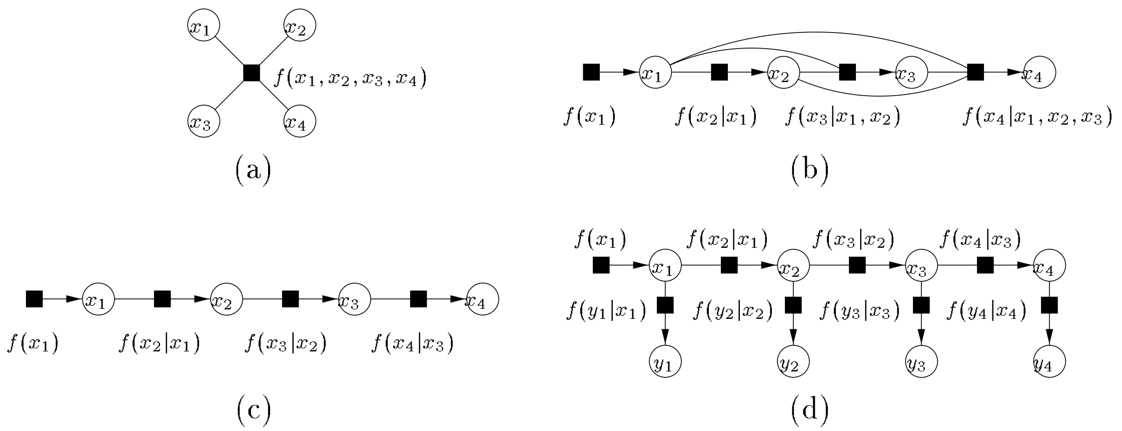


Figure 9: Factor graphs for probability distributions: (a) the trivial factor graph, (b) the chain-rule factorization, (c) a Markov chain, (d) a hidden Markov model.

In general, since all variables appear as arguments of $f(x_n|x_1,\ldots,x_{n-1})$, the factor graph of Fig. 9(b) has no advantage over the trivial factor graph shown in Fig. 9(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. 9(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. 9(d). Hidden Markov models are widely used in a variety of applications; see, e.g., [33] for a tutorial emphasizing applications in signal processing.

The strong resemblance between the factor graphs of Fig. 9(c) and (d) and the factor graphs representing trellises (Figs. 6(b) and 7) is not accidental; trellises can be viewed as Markov models for codes.

Of course, 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).

Example 7. (*Markov random fields*)
A Markov random field (see, e.g., [21]) 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:

$$(\forall v \in V) \;\; p(v|V \backslash \{v\}) = p(v|n(v)), \tag{10}$$

where, as usual, $n(v)$ 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., [21, 32, 19, 18]). Kschischang and Frey [22] give a brief discussion of the use of MRFs to describe codes.

Recall that 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. (Indeed, some authors takes this as the defining property of an MRF.) What this means is that the distribution factors as

$$p(v_1, v_2, \ldots, v_N) = Z^{-1} \prod_{E \in Q} f_E(V_E) \tag{11}$$

where Z^{-1} is a normalizing constant, and each $E \in Q$ is a clique. For example (*cf.* Fig. 1), the MRF in Fig. 10(a) can be used to express the factorization

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

(Although there are other cliques in this graph, e.g., $\{v_1, v_2\}$, we assume that any additional factors are absorbed in one of the factors given; for example, a factor of $f(v_1, v_2)$ would be absorbed by $f_C(v_1, v_2, v_3)$.)

Clearly (11) 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 (11), 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 [22].)

In the opposite direction, a factor graph F that represents a joint probability distribution can be converted to a Markov random field via a component of the second higher power graph F^2 [22].

Let $F(S, Q)$ be a factor graph, and let F^2 be the graph with the same vertex set as F, with an edge between two vertices v and v' in F^2 if and only if there is there is a path

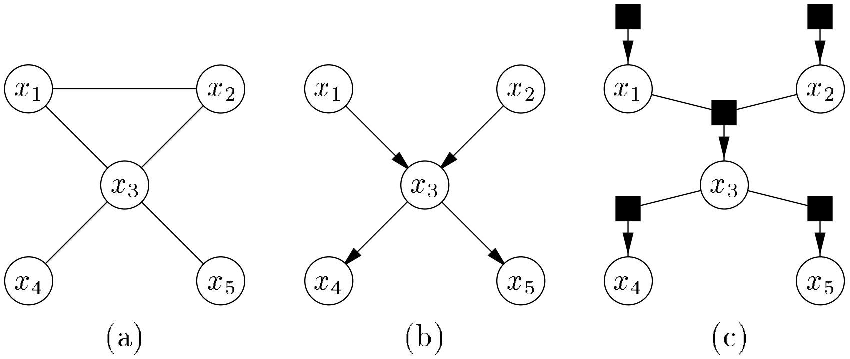


Figure 10: Graphical probability models: (a) a Markov random field, (b) a Bayesian network, (c) a factor graph.

of length two from v to v' in F. Since F is bipartite, F^2 decomposes into at least two components: one component, F_S^2, involving only variable nodes, and the other involving only function nodes. Since the arguments of a function node in F are connected by a path of length two in F, these arguments form a clique in F^2. For example, Fig. 10(a) shows F_S^2 for the factor graph F of Fig. 1.

The following theorem is proved in Appendix A.

Theorem 1 *If $F(S, Q)$ is a factor graph that represents a probability distribution as a product of non-negative factors, then F_S^2 is a Markov random field.*

This theorem can be interpreted as saying that, in a sense, a factor graph is the "square root" of a Markov random field.

Example 8. (*Bayesian networks*)
Bayesian networks (see, e.g., [31, 20, 11]) 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" [31], 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 [25], who independently re-discovered Gallager's earlier work on low-density parity-check codes [14] (including Gallager's decoding algorithm). More recently, at least two papers [22, 30] 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), the distribution represented by the Bayesian network assumes the form

$$p(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} p(v_i|\mathbf{a}(v_i)), \tag{12}$$

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.* (1)—Fig. 10(b) shows a Bayesian network that expresses the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = p(v_1)p(v_2)p(v_3|v_1, v_2)p(v_4|v_3)p(v_5|v_3). \tag{13}$$

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 (13) is shown in Fig. 10(c); *cf.* Fig. 1.

The arrows in a Bayesian network are often useful in modeling the "flow of causality" in practical situations; see, e.g., [31]. Provided that it is not required that a child variable take on some particular value, it is straightforward to *simulate* a Bayesian network, i.e., draw configurations of the variables consistent with the represented distribution. Starting from the variables having no parents, once values have been assigned to the parents of a particular variable x_i, one simply randomly assigns a value to x_i according the (local) conditional probability distribution $p(x_i|\mathbf{a}(x_i))$. Our arrow convention for factor graphs, noted earlier, and illustrated in Fig. 10(c), allows us to retain this advantage of Bayesian networks.

We note that factor graphs are more general than either Markov random fields or Bayesian networks, since they can be used to describe functions that are not necessarily probability distributions. Furthermore, factor graphs have the stronger property that every Markov random field or Bayesian network can be redrawn as a factor graph without information loss, while the converse is not true.

Example 9. (*Logic circuits revisited*)
Probability models are often obtained as an extension of behavioral models. One such case was given in Example 5. For another example, consider the logic circuit of Fig. 5, and suppose that $\{u_1, \ldots, u_4\}$ are random variables that assume particular configurations according to some *a priori* probability distribution. It is not difficult to see that, e.g., the joint probability mass function of u_1, u_2, and x_1 is given by

$$f(u_1, u_2, x_1) = f_{1,2}(u_1, u_2)[x_1 = u_1 \text{ AND } u_2]$$

where $f_{1,2}(u_1, u_2)$ is the joint probability mass function for u_1 and u_2. In this example, we will have

$$f(u_1, u_2, u_3, u_4, x_1, x_2, y) = f_{1,2,3,4}(u_1, u_2, u_3, u_4)g(u_1, u_2, u_3, u_4, x_1, x_2, y)$$

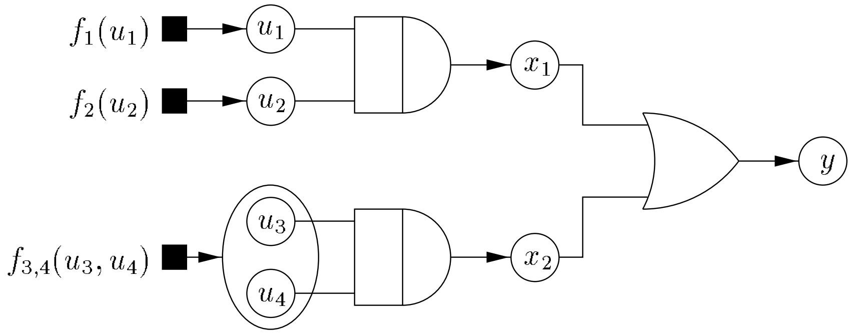


Figure 11: A factor graph representation for the joint probability mass function of the variables in the logic circuit example.

where $f_{1,2,3,4}(\cdot)$ is the joint probability mass function for the u_is and $g(\cdot)$ is the indicator function defined in (6). The factor graph corresponding to the case where $f(u_1, u_2, u_3, u_4)$ itself factors as $f_1(u_1)f_2(u_2)f_{3,4}(u_3, u_4)$ is shown in Fig. 11.

Fig. 11 includes arrows, showing the "flow of causality" from the inputs to the outputs of the logic gates. The pair of input variables u_3, u_4 are "clustered," i.e., treated as a single variable. (Variable clustering and other transformations of factor graphs is discussed in Section 6.) In general, for subsystems with a deterministic input/output relationship among variables, an output variable x_i can be viewed as a child variable having the input variables as parents; for each configuration of the parents, the corresponding conditional probability distribution assigns unit mass to the configuration for x_i consistent with the given input/output relationship.

As discussed, the arrows in Fig. 11 are useful in simulations of the distribution represented by the factor graph, as it is relatively straightforward to go from "inputs" to "outputs" through the graph. As we shall see, the sum-product algorithm will be useful for reasoning in the opposite direction, i.e., computing, say, conditional probability mass functions for the system inputs (or hidden variables) given the observed system output. For example, given models for "faults" in the various subsystems, we may be able to use the sum-product algorithm to reason about the probable cause of an observed faulty output.

2.3 Further Examples

We now give a number of further examples of factor graphs that might be used in a variety of fields, including artificial intelligence, neural networks, signal processing, optimization, and coding theory.

Example 10. (*Computer vision and neural network models*)
Graphical models have found an impressive place in the field of neural network models of perception [18, 17, 9, 12]. (See [11] for a textbook treatment.) Traditional artificial neural networks called "multilayer perceptrons" [34] treat perceptual inference as a function approximation problem. For example, the perceptron's objective might be to predict the relative shift between two images, providing a way to estimate depth. Initially, the perceptron's parameters are set to random values and the perceptron is very bad at predicting depth. Given a set of training images that are *labeled* by depth values, the perceptron's parameters can be estimated so that it can predict depth from a pair of images.

In the more realistic "unsupervised learning" situation, depth labels are not provided, but the neural network is supposed to extract useful structure from the data. One fruitful approach is to design algorithms that learn efficient *source codes* for the data. The parameters of a probability model are adjusted so as to minimize the relative entropy between the empirical data distribution and the distribution given by the model.

The graphical model (factor graph) framework provides a way to specify neurally plausible probability models. In a neural network factor graph, we can associate one local function with each variable such that the local function gives the probability of the variable given the activities of its local input variables. For example, the probability that a binary variable x_i is 1, given its binary inputs $\{x_j : j \in I_i\}$ may take the form,

$$P(x_i = 1|\{x_j : j \in I_i\}) = 1/(1 + \exp[-\textstyle\sum_{j\in I_i} w_{ij}x_j]),$$

where w_{ij} is the weight on the edge connecting x_i and x_j.

In the simplest case, we create one variable for each input variable. However, we can also introduce unobserved "hidden" or "latent" variables into the graphical model. Once the model is trained so that the marginal distribution over the visible variables is close to the empirical data distribution, we hope the hidden variables will represent useful features.

Although the details of these models and learning algorithms fall beyond the scope of this paper, we present here a brief example. See [13] for the details of a model and a learning algorithm for real-valued variables. In this example, we discuss a binary version of this problem for clarity. Each image in the pair is one-dimensional and contains 6 binary pixels. The training data is generated by randomly drawing pixel patterns for the first image and then shifting the image one pixel to the right or left (with equal probability) to produce the second image. Fig. 12(b) shows 4 examples of these image pairs, with the pair of images placed to highlight the relative shift.

Fig. 12(a) shows the factor graph structure that can be *learned* from examples of these image pairs. Initially, each variable was connected to all variables in adjacent layers and the parameters were set to random values. After learning, the weights associated with

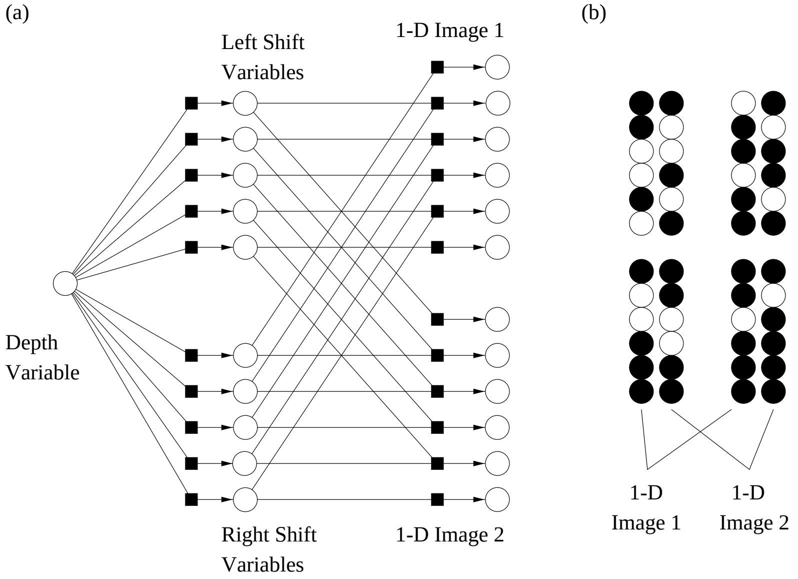


Figure 12: (a) The factor graph for a neural-network model of depth perception. (b) Examples of training cases for the simplified depth estimation problem.

unneeded edges (not shown) become close to zero. According to the remaining edges, each middle-layer variable represents the dependence between a pixel from the first image and another pixel from the second image. Also, the middle-layer variables can be separated into two groups: in one group, each variable links a pixel from one image to the pixel on its left from the other image; in the other group, right-shift relationships are represented. The dependencies between the middle-layer variables are captured by a “depth variable”. For one depth activation, all the middle-layer variables in one group are shut off, whereas the middle-layer variables in the other group may or may not be active. Without labeled training data, the unsupervised learning algorithm can extract an architecture that can predict depth.

One challenging research problem is the development of algorithms that can learn structures like the one described above, but on a much larger scale. In real-world data, the images are 2-dimensional, there is a much larger number of pixels, the depth varies across the image, and the network must interact with other feature detectors and other sensory modalities.

Example 11. (*The DFT kernel*)
Next, we provide an example from the field of signal processing, where a widely used tool for the analysis of discrete-time signals is the discrete Fourier transform. 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. Recall that the discrete Fourier transform of $\mathbf{w}$ is the

complex-valued N-tuple $\mathbf{W} = (W_0, \ldots, W_{N-1})$ where

$$W_k = \sum_{n=0}^{N-1} w_n \Omega^{-nk}, \quad k = 0, 1, \ldots, N-1. \tag{14}$$

Consider now the case where N is a power of two, e.g., $N = 8$ for concreteness. We express variables n and k in (14) 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. 13.

We observe that

$$W_k = W_{4y_2+2y_1+y_0} = \sum_{x_0} \sum_{x_1} \sum_{x_2} g(x_0, x_1, x_2, y_0, y_1, y_2) \tag{15}$$

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. In Section 6, we will see that such factor graph representations can lead to fast Fourier transform (FFT) algorithms.

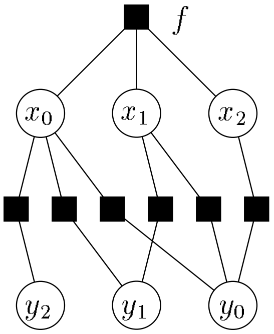


Figure 13: A factor graph for the discrete Fourier transform kernel function.

Example 12. (*Nonserial dynamic programming*)
In the field of optimization, in their formulation [8] of so-called nonserial dynamic programming problems, Bertelé and Brioschi consider functions that are the *additive* composition of local functions. These types of problems fit naturally in the factor graph

framework, provided that we view the 'multiplication' operation in R as real-valued addition. If, as in [8], we are interested in minima and maxima of the global function, we will take as a summary operator the 'min' or 'max' operation.

It is interesting to observe the close relationship between the "interaction graph" of a nonserial dynamic programming problem [8, p. 4] and the graph corresponding to a Markov random field (see example 7): they are really the same object defined with respect to a different binary operation: function addition in the nonserial dynamic programming case and function multiplication in the Markov random field case. As in the the Markov random field case, interaction graphs can be recovered from the second higher power of the factor graph.

Example 13. (*Weight enumerating functions*)
Finally, we present a non-probabilistic example from coding theory. Let C be a linear code of length n and let $[(x_1, \ldots, x_n) \in C]$ be the binary indicator function for membership in C. Denote the Hamming weight of a codeword $\mathbf{x} \in C$ by $wt(\mathbf{x})$. Let z be an indeterminate. Then the function

$$g(x_1, \ldots, x_n) = [(x_1, \ldots, x_n) \in C] \prod_{i=1}^{n} z^{[x_i \neq 0]}$$

takes on the value $z^{wt(\mathbf{x})}$, whenever $\mathbf{x}$ is a valid codeword, and zero otherwise. We can think of g as a modified indicator function that not only indicates whether $\mathbf{x}$ is a valid codeword, but also returns the Hamming weight of $\mathbf{x}$ (as the z-exponent) whenever $\mathbf{x}$ *is* a valid codeword. If we define a summary operator in terms of the usual addition of polynomials in z, then $g \downarrow X_\emptyset$ is the Hamming weight enumerator for C.

A factor graph representation for g can be obtained by augmenting a factor graph representation for the code membership indicator function with the local functions $z^{[x_i \neq 0]}$, for $i = 1, \ldots, n$. For the code of example 1, this would lead to a factor graph with the structure of Fig. 8(b).

3 Function Cross-Sections, Projections, and Summaries

As in Section 1, let X_S be a collection of variables indexed by a set S, let A_S be a configuration space for these variables, and let $g\colon A_S \to R$ be a function of these variables into a commutative semigroup R. If g factors as in (2), let $F(S, Q)$ be the corresponding factor graph.

3.1 Cross-Sections

Let $E \subset S$ be a subset of the variable index set S, let X_E be the set of variables indexed by E, and let a_E be a particular subconfiguration with respect to E, viewed, as defined in Section 1, as the multiple variable assignment $X_E = a_E$. Every such assignment yields a function *cross-section*

$$g(X_{S\setminus E} \| X_E = a_E).$$

For example if $g(x_1, x_2, x_3) = x_1x_2x_3$, then $g(x_1, x_2 \| x_3 = 5) = 5x_1x_2$. If $E = S$, then $g(X_\emptyset \| X_E = a_E)$ is the constant $g(a_E)$.

A factor graph for the cross-section $g(X_{S\setminus E} \| X_E = a_E)$ is obtained from a factor graph for g simply by

1. replacing each local function having an argument x_i for any $i \in E$ with its corresponding cross-section; and
2. omitting variable nodes indexed by E and any edges incident on them.

For example, let $g(x_1, x_2, x_3) = a(x_1, x_2)b(x_2, x_3)$, with factor graph shown in Fig. 14(a). A factor graph for the cross-section $g(x_1, x_3 \| x_2 = a_2) = a(x_1 \| x_2 = a_2)b(x_3 \| x_2 = a_2)$ is shown in Fig. 14(b).

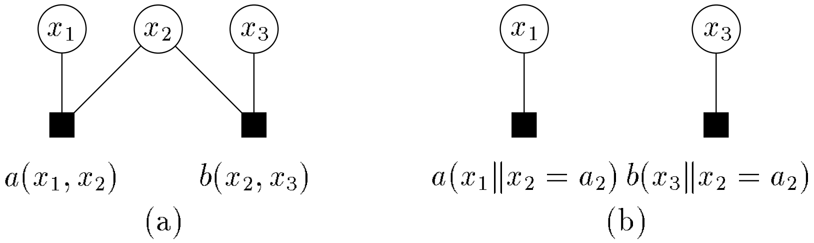


Figure 14: Factor graphs for (a) $g(x_1, x_2, x_3) = a(x_1, x_2)b(x_2, x_3)$, and (b) the cross-section $g(x_1, x_3 \| x_2 = a_2) = a(x_1 \| x_2 = a_2)b(x_3 \| x_2 = a_2)$.

If $g(X_S)$ is a probability mass or density function then its cross-section $g(X_{S\setminus E} \| X_E = a_E)$ is linearly proportional to the conditional probability mass or density function for the variables indexed by $S \setminus E$, given the particular subconfiguration a_E for the variables indexed by E.

Two variables x_i and x_j contribute independently to the cross-section $g(x_{S\setminus E} \| X_E = a_E)$ if and only if x_i and x_j are contained in distinct components of the factor graph for the cross-section. Thus, if x_i and x_j are in the *same* component of the factor graph for g to begin with, they contribute independently to the cross-section if and only if the vertices of E *separate* x_i and x_j in the factor graph for g. (In the probability case, we

x	y	z	$g(x,y,z)$
0	0	0	$g(0,0,0)$
0	0	1	$g(0,0,1)$
0	1	0	$g(0,1,0)$
0	1	1	$g(0,1,1)$
1	0	0	$g(1,0,0)$
1	0	1	$g(1,0,1)$
1	1	0	$g(1,1,0)$
1	1	1	$g(1,1,1)$

(a)

x	$g(x,y,z)$
0	$g(0,0,0)$
0	$g(0,0,1)$
0	$g(0,1,0)$
0	$g(0,1,1)$
1	$g(1,0,0)$
1	$g(1,0,1)$
1	$g(1,1,0)$
1	$g(1,1,1)$

(b)

Table 1: (a) A function viewed as a relation. (b) The projection on its first coordinate.

would say that the corresponding random variables are conditionally independent given those random variables indexed by E.)

If variable subsets X_E and $X_{S\setminus E}$ contribute independently to a function $g(X_S)$, then, up to scale, the cross-section $g(X_{S\setminus E}\|X_E = a_E)$ for every configuration x_i of X_I "looks the same." More precisely, there exists a function $f(X_{S\setminus I}) \to R$ such that every cross-section $g(x_{S\setminus I}\|X_I = x_i)$ is a scalar multiple of $f(X_{S\setminus I})$.

3.2 Projections and Summaries

A function $g\colon A_S \to R$ can be viewed as a relation (i.e., as a particular subset G of $A_S \times R$) in which, for every $w \in A_S$, there is a unique element $(w, g(w))$ in G. The set A_S is the domain of G and R is the codomain.

Consider now the projection of G on some set of coordinates in its domain. Unlike a cross-section, the projection of a function results in a relation that in general is *not* a function. For example, the projection of the function shown in Table 1(a) on its first coordinate (x) is the relation shown in Table 1(b). This relation is not in general a function $f\colon A_x \to R$ since a particular $x \in A_x$ can stand in relation to more than one element of R. Nevertheless, such a projection can be converted to a function by introducing an operator that "summarizes" in some sense the multiple function values associated with each configuration of the relation domain.

For example, if g is real-valued, one way of summarizing the multiple function values associated with each value of x would be to take their sum, i.e., define a new function $h(x) = \sum_{y,z} g(x,y,z)$. If g were a probability mass function, then h would be a marginal probability mass function. Another possible summary operation would be to take the maximum of the multiple function values, i.e., define $h(x) = \max_{y,z} g(x,y,z)$. Clearly a large number of summary operators are possible.

We first introduce a notation for such an operator. If $g(X_C)\colon A_C \to R$ is a function and $B \subset C$, we denote by

$$g(X_C) \downarrow X_B\colon A_B \to R$$

the new function—called the *marginal* or *summary* of g for X_B—obtained by "summarizing," for each fixed configuration of the variables in X_B, the multiple function values associated with all possible configurations of those arguments *not* in X_B. For convenience we will often refer the *variables* in $X_C \setminus X_B$ as being summarized; by this we mean, of course, the summary of the corresponding function values. Thus, if g is the function of Table 1, the marginal of g for $\{x_1\}$, i.e., the summary of the multiple function values in Table 1(b), would be denoted as $g(x_1, x_2, x_3) \downarrow \{x_1\}$, or simply $g(x_1, x_2, x_3) \downarrow x_1$, and we would refer to x_2 and x_3 as having been summarized.

Although it would seem that a summary operator should apply only to the case where $B \subset C$, we extend it to the general case where B and C are arbitrary subsets of some index set S by defining

$$g(X_C) \downarrow X_B\colon A_{B\cap C} \to R$$

as

$$g(X_C) \downarrow X_B = g(X_C) \downarrow X_{B\cap C}. \tag{16}$$

In other words, the marginal of $g(X_C)$ for X_B is the function obtained by summarizing those arguments of g that are not in X_B, i.e., by summarizing the variables indexed by $C \cap (S \setminus B)$. Note that $g(X_C) \downarrow X_\emptyset$ is a summary of *all* of the function values, i.e., a summary of $g(X_C)$ itself.

We now consider the properties that a summary operator should possess. For every nonempty subset B of the index set S, let R^{A_B} denote the set of all functions from A_B to R. We extend this notation to the case where B is empty by defining $R^{A_\emptyset} = R$. Suppose we have a family of mappings such that for every pair $\{B, C\}$ of subsets of S, with $C \subset B$, there is a mapping $\downarrow A_C\colon R^{A_B} \to R^{A_C}$ in this family that associates an element of R^{A_C} (written in suffix notation as $g(X_B) \downarrow A_C$) with each element $g(X_B)$ of R^{A_B}. We extend this family of mappings to arbitrary B and C be defining $\downarrow A_C\colon R^{A_B} \to R^{A_{B\cap C}}$ as in (16), and denote the result by $\downarrow$.

A family of mappings $\downarrow$ as defined in the previous paragraph will be called a *summary operator* if, for all R-valued functions f and g and for all variable index sets B, C, and D, the following axioms are satisfied:

1. $B \subset C \Rightarrow f(X_D) \downarrow X_B = [f(X_D) \downarrow X_C] \downarrow X_B$;
2. $B \subset C \Rightarrow [f(X_B) \cdot g(X_D)] \downarrow X_C = f(X_B) \cdot [g(X_D) \downarrow X_C]$;

3. $B \cap C = \emptyset \Rightarrow [f(X_B) \cdot g(X_C)] \downarrow X_D = [f(X_B) \downarrow X_D] \cdot [g(X_C) \downarrow X_D]$.

These axioms are modeled on the properties of marginal functions in probability theory.

Axiom 1 implies that marginals can be obtained in a sequence of "stages" of ever more restrictive summaries, and that any such sequence will yield the same function. Thus, for the example of Table 1,

$$\begin{aligned} g(x_1, x_2, x_3) \downarrow x_1 &= (g(x_1, x_2, x_3) \downarrow \{x_1, x_2\}) \downarrow x_1 \\ &= (g(x_1, x_2, x_3) \downarrow \{x_1, x_3\}) \downarrow x_1, \end{aligned}$$

i.e., we can summarize x_3 and then x_2 or *vice versa* and obtain the same marginal function in either case. Axioms 2 and 3 describe how summary operators behave with respect to function products. If $B \subset C$, then for every fixed configuration of the variables of X_C, the function $f(X_B)$ is a constant; Axiom 2 states that this constant can be factored out from the marginal function. Axiom 3 says that if two functions do not "interact," i.e., if they have no arguments in common, then the marginal of the product is the product of the marginals. This axiom extends easily to the case of the product of more than two functions with disjoint arguments.

The following Lemma will be important later.

Lemma 1 *If X_A and X_B are disjoint and do not contain x, then*

$$[f(x, X_A) g(x, X_B)] \downarrow x = [f(x, X_A) \downarrow x] \cdot [g(x, X_B) \downarrow x].$$

Proof: We write

$$\begin{aligned} [f(x, X_A) g(x, X_B)] \downarrow x &= [(f(x, X_A) g(x, X_B)) \downarrow \{x\} \cup X_A] \downarrow x \\ &= [f(x, X_A)(g(x, X_B) \downarrow \{x\} \cup X_A)] \downarrow x \\ &= [f(x, X_A)(g(x, X_B) \downarrow x)] \downarrow x \\ &= [f(x, X_A) \downarrow x] \cdot [g(x, X_B) \downarrow x] \end{aligned}$$

where the first equality follows by applying Axiom 1, the second equality follows from Axiom 2, and the third equality follows from the assumption that X_A and X_B are disjoint, and the fourth equality follows from another application of Axiom 2. ▮

This lemma can clearly be extended to the analogous situation involving the product of more than two functions with arguments intersecting (pairwise) in $\{x\}$. For example, if X_A, X_B and X_C are pairwise disjoint and do not contain x, then

$$\begin{aligned} [f(x, X_A) g(x, X_B) h(x, X_C)] \downarrow x &= [(f(x, X_A) g(x, X_B)) \downarrow x] \cdot [h(x, X_C) \downarrow x] \\ &= [f(x, X_A) \downarrow x] \cdot [g(x, X_B) \downarrow x] \cdot [h(x, X_C) \downarrow x]. \end{aligned}$$

3.3 Summary Operators via Binary Operations

When all symbol alphabets A_i, $i \in S$, are finite, we will often define a summary operator in terms of a commutative, associative binary operation in R. We denote this binary operation by '+', thinking of "sum" for "summary." As in [1, 2, 23, 28, 37, 38], we will insist that the summary operator + satisfy the distributive law in R, i.e., for all $x, y, z \in R$,

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z), \tag{17}$$

i.e., we will require that $(R, +, \cdot)$ form a semiring, where $\cdot$ is the multiplication operation usually denoted in this paper by juxtaposition.

For arbitrary subsets B and C of S, we define the marginal function $g(X_B) \downarrow X_C$ as

$$g(X_B) \downarrow X_C = \sum_{x_i \in A_i : i \in B \cap (S \setminus C)} g(X_B), \tag{18}$$

i.e., by summing over all configurations of those variables *not* indexed by C. When $B \cap (S \setminus C)$ is empty, we take $g(X_B) \downarrow X_C = g(X_B)$. This definition clearly gives us a family of mappings that associate with each element of R^{A_B} and element of $R^{A_{B \cap C}}$. For example, the marginal of g with respect to x_1 for the function of Table 1(b) would be computed as

$$g(x_1, x_2, x_3) \downarrow x_1 = \sum_{y \in A_y} \sum_{z \in A_z} g(x, y, z).$$

From the distributive law (17) and the commutativity of the product operation in R, it is clear that the summations in (18) can be performed in an arbitrary order, and hence Axiom 1 for summary operators is satisfied. Furthermore, from the distributive law (17) it follows that Axiom 2 is satisfied. Finally, if f and g have no arguments in common, then the distribute law in R once more implies that Axiom 3 is satisfied. Therefore, we have the following theorem.

Theorem 2 *The family of mappings defined in (18) is a summary operator.*

An important example of a summary operator arises in the case that R is the set of reals with the usual multiplication, and + is taken as ordinary real addition. When $g(x, y, z)$ is the joint probability mass function of random variables $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ then $g(x, y, z) \downarrow x$ is the *marginal* probability mass function for $\mathbf{X}$.

More generally, if $g(x, y, z)$ represents a probability density function for the jointly continuous random variables $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$, by replacing the sums in (18) with integrals, we can define a family of mappings that take a function to the appropriate marginal function, so that, e.g.,

$$g(x,y,z) \downarrow x = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x,y,z)\, dy\, dz.$$

It is a straightforward exercise to verify that this definition gives a family of mappings that satisfies the axioms required of summary operators.

Another possible summary operator when R is the reals is obtained from the "min-sum" semiring, where $\cdot$ is ordinary addition in R and $+$ is the 'min' operator, i.e., $x + y \triangleq \min(x,y)$. In this case, $g(x,y,z) \downarrow x$ is the minimum of $g(x,y,z)$ taken over all configurations agreeing at x.

4 The Sum-Product Algorithm

Having introduced the concept of a factor graph in Section 1 and the concept of a summary operator in Section 3, in this section we combine these concepts by describing a generic algorithm—the *sum-product algorithm*—that operates in a factor graph $F(S,Q)$ via a sequence of "local" computations. These computations follow a single conceptually simple rule, combining the product and summarization operations. The results of these local computations are passed as *messages* along the edges of the factor graph.

In general, a large variety of possible message-passing *schedules* in a factor graph F are possible. When F is a tree, we will present a family of schedules called generalized forward/backward (GFB) schedules, that cause exactly one message to pass in either direction on every edge of the tree. We will prove that the result of this computation is the set of marginal functions $g(X_S) \downarrow x_i$ for all $i \in S$, or for specific $i \in E \subset S$.

The sum-product algorithm can also be applied to factor graphs with cycles. Because of the cycles in the graph, an "iterative" algorithm with no natural termination will result. 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, as we shall make clear later in the paper, 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.

As usual, let X_S be a collection of variables indexed by a set S, let A_S be a configuration space for these variables, let $g\colon A_S \to R$ be a function of these variables into a commutative semigroup $(R,\cdot)$, and let $\downarrow$ denote a summary operator for R-valued functions. Finally, suppose g factors as in (2) for some collection Q of subsets of S, and let $F(S,Q)$ be the corresponding factor graph.

4.1 Computation by Message-Passing

The sum-product algorithm is perhaps best described by imagining that there is a processor at each node of the underlying factor graph F, and that the edges in F represent channels by which these processors may communicate by sending "messages." For us, messages will always be R-valued *functions* or descriptions thereof.

For example, if x is a (binary) variable taking on values in $\{0, 1\}$, the real-valued function $f(x)$ can be described by the vector $(f(0), f(1))$, and this vector could be passed as a message along some edge in F. Equivalently, the vector $(f(0) + f(1), f(0) - f(1))$ contains the same information, and hence, if convenient, could also be taken as a description of $f(x)$. If $f(x)$ is a probably mass function so that $f(0) + f(1) = 1$, then a number of scalar-valued function descriptions (e.g., $f(0)$, $f(1)$, $f(0) - f(1)$, $f(0)/f(1)$, $\log(f(0)/f(1))$, etc.) would, as convenient, provide adequate description of f. If x is continuous-valued, and $f(x)$ is drawn from a parameterized set (e.g., a probability density function in an exponential family), then a vector containing the function parameters could be used as a description of f.

We will often have cause to form the products of messages. By this we mean, of course, products of the corresponding functions, not products the actual messages themselves. Of course, if $f(x)$ and $g(x)$ are two functions of the same discrete variable x, and these functions are described using ordered vectors containing the function values, then the vector description of the product of f and g *is* the component-wise product of the vector descriptions of f and g.

4.2 The Sum-Product Update Rule

Deferring, for the moment, the question of algorithm initialization, we describe now the one simple computational rule followed by the sum-product algorithm.

> **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.

Note that the message sent on an edge $\{x, f\}$, where x is a variable node and x is a function node, is always a function of the variable x associated with that edge. Let us denote by $\mu_{v \to w}(x_{\{v,w\}})$ the message sent from node v to node w by the sum-product algorithm. Then, as illustrated in Fig. 15, the message computations performed by the sum-product algorithm can be expressed as follows:

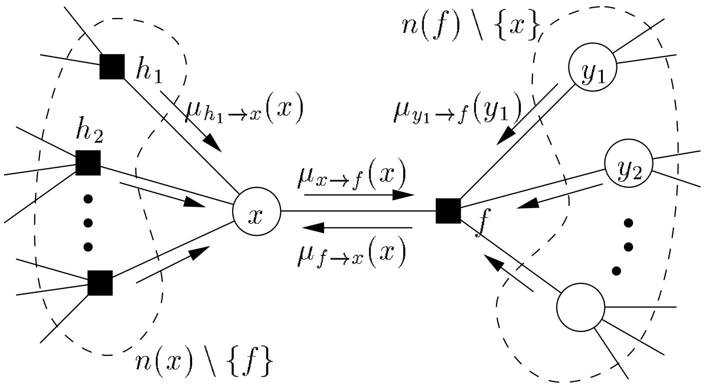


Figure 15: A factor graph fragment, showing the sum-product algorithm update rules.

variable to local function:

$$\mu_{x \to f}(x) = \prod_{h \in n(x) \setminus \{f\}} \mu_{h \to x}(x) \tag{19}$$

local function to variable:

$$\mu_{f \to x}(x) = \left(f(X_{n(f)}) \prod_{y \in n(f) \setminus \{x\}} \mu_{y \to f}(y) \right) \downarrow x \tag{20}$$

The update rule at a variable node x takes on the particularly simple form given by (19) 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 (20) in general involves nontrivial function multiplications, followed by an application of the summary operator.

4.3 Message Passing Schedules

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, 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.

Although not necessary in a practical implementation, we will make the assumption that message passing is synchronized with a global discrete-time clock. We will assume that only one message may pass on any edge in any given direction during one clock tick, and that this message *replaces* any previous message passed on that edge in the given 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.

A message passing *schedule* for the sum-product algorithm in a factor graph $F(S, Q)$ is a sequence $L = \{L_0, L_1, \ldots\}$, where $L_i \subset (S \times Q) \cup (Q \times S)$ is a set of edges, now considered to be directed, over which messages pass at time i. If $(v, w) \in L_i$, then, according to the schedule L, node v sends a message to node w at time i.

Obviously a wide variety of message passing schedules are possible. For example, the schedule in which $L_i = (S \times Q) \cup (Q \times S)$ for all i, is called the *flooding schedule* [22]. The flooding schedule calls for a message to be passed on each edge in each direction at each clock tick. If, for all i, L_i is a singleton, i.e., $|L_i| = 1$, the resulting schedule—called a *serial schedule*—calls for only one message (in the entire graph) to be passed during each clock tick. Despite the generality of possible schedules, we can make a few observations.

We say that a vertex v has a message *pending* at an edge e if the message that v can send on e is (potentially) different from the previous message sent on e. For example, variable nodes initially have no messages pending, since they would initially only send a unit message, and this is exactly what is initially assumed to be sent. Function nodes, on the other hand, can send a description of the local function (appropriately summarized) on any edge. In general this summary is not a unit function, and hence, before any messages have been passed, function nodes have messages pending on each incident edge.

When a message is received at a node, this will in general cause a change in the values of the messages to be sent on all *other* edges incident on that node. Hence the receipt of a message on an edge e at a node v causes v to have a message pending on all edges incident on v, other than e. Of course, the receipt of a message at a leaf node creates no messages pending, since there are no edges other than e. Thus leaf nodes *absorb* pending messages, whereas non-leaf nodes *distribute* pending messages.

Only pending messages need to be sent at any clock tick since, by definition, only pending messages have the potential to be different from the previous message sent on a given edge. However, it is not necessary for *every* pending message to be sent at a given time. We call a message-passing schedule *nowhere idle* if at least one pending message is sent at each clock tick.

The flow of pending messages in any given schedule can be visualized using diagrams like those shown in Fig. 16, in which a pending message is shown as a dot near the given edge. The transmission of a message is indicated by attaching an arrow to the dot. The flow of time is also indicated, with messages sent at non-negative integer times t.

Fig. 16 shows two possible message-passing schedules for the same factor graph. Only pending messages are shown. In Fig. 16(a), the *flooding schedule* (in which all pending messages are sent during each clock tick) is shown, while in Fig. 16(b), a *two-way schedule*, defined as a schedule in which exactly one message passes in each direction over a given edge, is shown. Explicitly, the schedule in L in Fig. 16(b) is given by $L = \{L_0, L_1, L_2\}$, where

$$\begin{aligned} L_0 &= \{(f_1, x_3), (f_2, x_3), (f_3, x_3)\}, \\ L_1 &= \{(x_3, f_1), (x_3, f_2), (x_3, f_3)\}, \\ L_2 &= \{(f_1, x_1), (f_1, x_2)\}; \end{aligned}$$

where these labels are shown in Fig. 16(c).

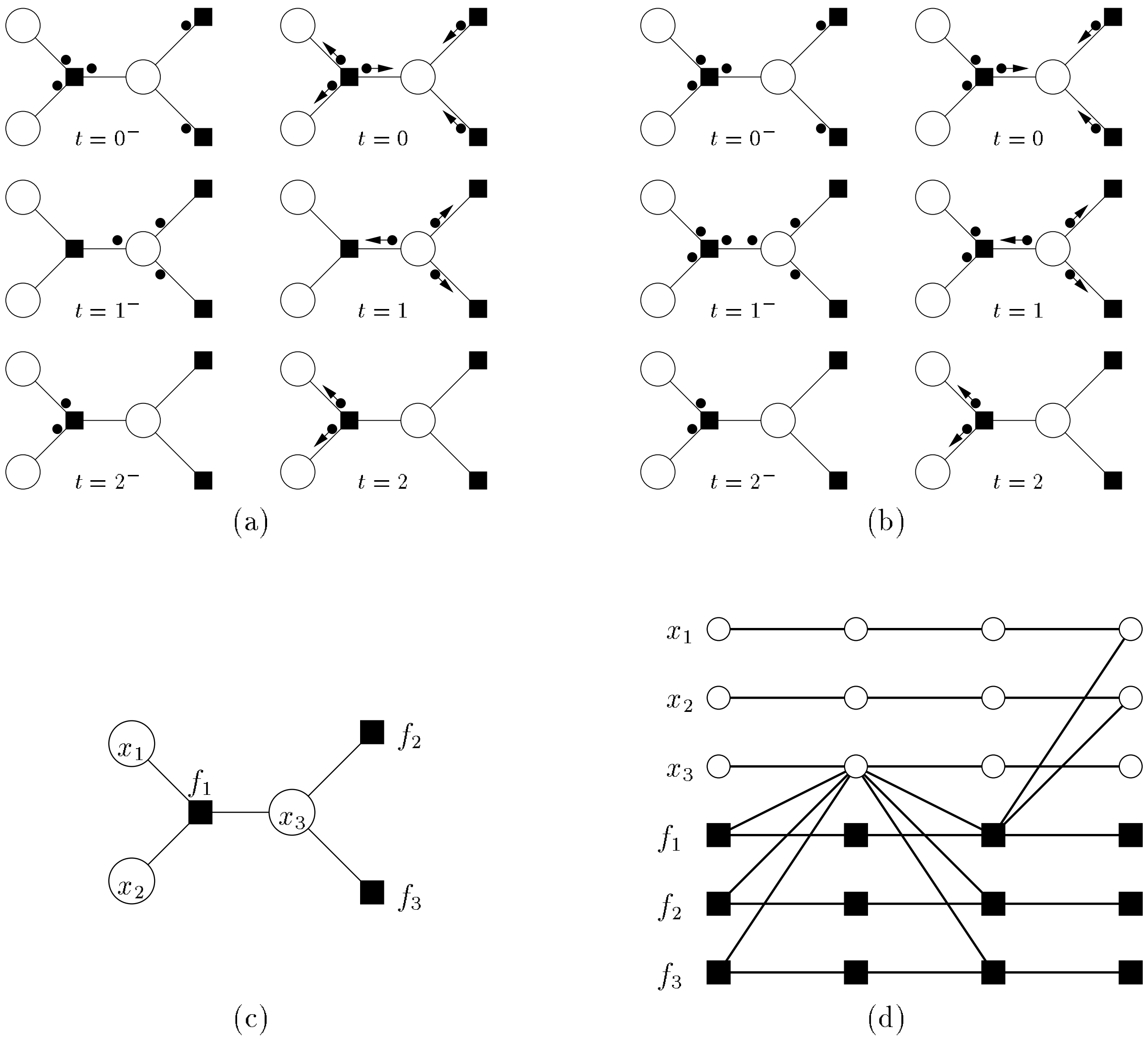


Figure 16: Two message passing schedules for the factor factor graph (c): in (a), the flooding schedule for pending messages is shown; in (b) a two-way schedule is shown. In (d) a computation trellis for schedule (b) is shown.

A given schedule L for a factor graph F can also be visualized via a *computation trellis*, like the one shown in Fig. 16(d). Here, the trellis vertices V at each depth correspond to the vertices of the factor graph. The edges in the ith trellis section are precisely

$L_i \cup \{(v,v) : v \in V\}$, and are considered to be directed from left to right. In the ith trellis section, the left vertices are considered to be at depth i and the right vertices at depth $i+1$.

Because leaf nodes absorb pending messages, if the sum-product algorithm uses a nowhere idle schedule in a finite tree, then it will eventually arrive at a state in which there are no nodes with messages pending. However, this does not happen in a graph with cycles. For example, Fig. 17 shows the flow of messages for the flooding schedule in a factor graph with a single cycle. Since the state of pending message in the graph the same at time 0^-, 2^-, 4^-, etc., it is clear that the algorithm never terminates in a situation in which there are no messages pending.

In fact, in any graph with a cycle, *no* message-passing schedule will result in transmission of *all* pending messages, since transmission of a message on any edge of the cycle will eventually cause that message to reach the originating node, triggering that node 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 i, (or, perhaps, for i in a selected subset of S) 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. As we shall see, when F is a finite tree, and L is a GFB schedule, the terminating computation at x_i yields the marginal function $g(X_S) \downarrow x_i$.

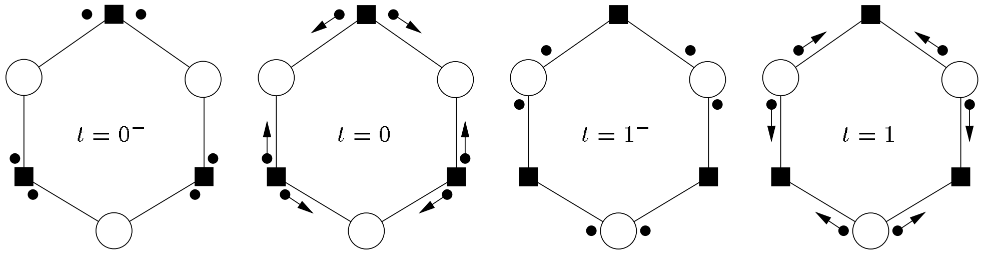


Figure 17: The flooding schedule for a factor graph with a single cycle.

4.4 The Sum-Product Algorithm in a Finite Tree

4.4.1 The Articulation Principle

Let us assume now that F is a finite tree. As discussed in Section 1, the graph obtained from F by cutting any edge $\{v,w\}$ is the union of two components, one denoted $F_{w \to v}$ (containing v, but not w) and the other denoted $F_{v \to w}$ (containing w but not v). Intuitively, in the operation of the sum-product algorithm, all information about the local functions in $F_{w \to v}$ must flow (from v to w) over the edge $\{v, w\}$, and similarly in the opposite direction. Of course, since we will in general be interested in function summaries, the information about $F_{w \to v}$ sent from v to w can in general be summarized for $x_{\{v,w\}}$ and the variables involved in $F_{v \to w}$.

We refer to this intuitive idea as the *articulation principle*: the message $\mu_{v \to w}$ sent from v to w must *articulate* (i.e., encapsulate or summarize) for the variables attached to $F_{v \to w}$ the product of the local functions in $F_{w \to v}$. Formally, we would like to ensure that

$$\mu_{v \to w} = \left(\prod_{E \in Q_{w \to v}} f_E(X_E) \right) \downarrow X_{v \to w},$$

where $Q_{w \to v}$ is the set of function nodes in $F_{w \to v}$ and $X_{v \to w}$ is the set containing $x_{\{v,w\}}$ and the variables of $F_{v \to w}$, i.e.,

$$X_{v \to w} = \bigcup_{E \in Q_{v \to w}} X_E.$$

The articulation principle is closely related to the notion of "state" or "sufficient statistic:" all that needs to be known about about the product of the functions on one side of the cut by the processors on the other side of the cut is captured (articulated) in the message sent across the cut.

Our first observation, when F is a tree, is that the set of variables that appear as arguments in the functions on one side of the cut and the set of variables that appear as arguments in the functions on the other side of the cut only have $x_{\{v,w\}}$ in common, i.e.,

$$X_{v \to w} \cap X_{w \to v} = x_{\{v,w\}}.$$

From (16) it follows that

$$\left(\prod_{E \in Q_{w \to v}} f_E(X_E) \right) \downarrow X_{v \to w} = \left(\prod_{E \in Q_{w \to v}} f_E(X_E) \right) \downarrow x_{\{v,w\}}.$$

Hence, the message passed from v to w needs only to be a function of $x_{\{v,w\}}$, and we can write the articulation principle for trees as

$$\mu_{v \to w}(x_{\{v,w\}}) = \left(\prod_{E \in Q_{w \to v}} f_E(X_E) \right) \downarrow x_{\{v,w\}}. \tag{21}$$

4.4.2 Generalized Forward/Backward Schedules

We now present a family of schedules for trees called generalized forward/backward (GFB) schedules that will cause exactly one message to be passed in each direction along every edge of F. We shall prove that each message satisfies the articulation principle (21). A complexity analysis for the GFB schedules is given in Appendix B, for the special case in which all alphabets are finite and the summary operator is defined in terms of a binary operation.

Of course, the basic update equations (19) and (20) will be followed, but we add the proviso that a node v may send a message on an edge e only if it has received messages on all *other* edges incident on v. Clearly, if $\partial(v) = d$, v can *send* a message once it has *received* at least $d - 1$ messages. We refer to this as the *all-but-one rule.*

According to the all-but-one rule, a leaf node (and only a leaf node) can send a message initially, and hence the GFB schedules begin at the leaf nodes. If v is a leaf variable node, the message sent is the unit function, while if v is a leaf function node, the message sent is (a description of) the function $v(X_{n(v)})$. More formally,

initialization: (at each leaf variable node x and each leaf function node f)

$$\begin{aligned} \mu_{x\to n(x)}(x) &= 1 \\ \mu_{f\to n(f)}(X_{n(f)}) &= f(X_{n(f)}). \end{aligned} \tag{22}$$

Suppose, after a message is sent from a leaf node v, that v (and the edge incident on v) is deleted from F, resulting in a graph F'. As we observed in Section 1, F' is still a tree. Furthermore, any leaf node in F' is in a position to send a message, since it will have received messages on all but one edge. Thus we can choose one of the leaf nodes in F', compute the message it should send on its outgoing edge, send the message, and delete the node and its edge from F', to form a new graph F'', and so on. Two basic invariant properties of the sequence of graphs constructed in this way are: (1) leaf nodes are always in a position to send a message, and (2) no messages have been sent on any edge. Clearly this process can continue until just a single node remains. Since at each stage the graph is a tree of more than one node, there are always at least two leaf nodes in a position to send a message, so this process will not stall.

At the point when just a single node v remains, a message will have passed in one direction over each edge in the original graph F. Furthermore, v will have received messages on *all* of its edges. We now re-construct F starting at v. We can choose any neighbor w of v (in F), and adjoin v and the edge $\{v, w\}$ to form a new graph that we (again) call F', and send a message from v to w. At this point, both v and w will have received messages on *all* of their edges. We then adjoin any neighbor of v and w (and the corresponding edge) to form the graph F'', sending a message to the newly added vertex. This vertex, too, will then have received messages on *all* of its edges. Continuing in this manner, we can always adjoin a neighboring node to the graph until, finally, all of the nodes of F have been added. Two basic invariant properties of this sequence of graphs are: (1) every node will have received message on all of its edges, and (2) exactly two messages—one in either direction—will have been sent on every edge. Of course these properties will also be true of the last graph in the sequence, namely F itself.

The algorithm terminates, once F has been regenerated, by computing, for each i, (or, perhaps, for i in a selected subset of S) the product of the messages received at variable node x_i, or equivalently, the product of the messages sent and received on any single edge incident on x_i. More formally, we have

termination:

$$\begin{aligned} g(X_S) \downarrow x &= \prod_{f \in n(x)} \mu_{f \to x}(x) \\ &= (\mu_{x \to f}(x))(\mu_{f \to x}(x)) \text{ for any } f \in n(x). \end{aligned} \tag{23}$$

Note that, since F is finite, this algorithm terminates in a finite number of steps. Below, we will prove the correctness of the equality (23). However, first we must show that the messages passed during the operation of a GFB schedule satisfy the articulation principle (21). This is expressed in our main theorem.

Theorem 3 *If F is a finite tree and and the sum-product algorithm follows a generalized forward/backward schedule in F, then all messages satisfy the articulation principle, i.e,. for neighbors v and w in F,*

$$\mu_{v \to w}(x_{\{v,w\}}) = f_{w \to v}(X_{w \to v}) \downarrow x_{\{v,w\}}, \tag{24}$$

where $f_{w \to v}(X_{w \to v})$ is the product of the local functions in $F_{w \to v}$.

Proof: We proceed by induction on the order in which messages are sent in the sum-product algorithm. From (22), statement (24) is clearly true whenever v is a leaf node. If we can show that (24) is true for all outgoing messages whenever it is true for incoming messages at a node, then it will follow by induction that (24) is true for all messages. Designate $\{v, w\}$ as the outgoing edge at v and suppose that the equivalent of (24) is satisfied for all incoming messages at v, i.e.,

$$(\forall u \in n(v) \setminus w) \;\; \mu_{u \to v}(x_{\{u,v\}}) = f_{v \to u}(X_{v \to u}) \downarrow x_{\{u,v\}}.$$

If v is a variable node, $x_{\{u,v\}} = v$. For distinct $u_1, u_2 \in n(v)$, $X_{v \to u_1} \cap X_{v \to u_2} = \{v\}$; hence, from Lemma 1, we obtain

$$\begin{aligned} \mu_{v \to w}(v) &= \prod_{u \in n(v) \setminus w} (f_{v \to u}(X_{v \to u}) \downarrow v) \\ &= \left(\prod_{u \in n(v) \setminus w} f_{v \to u}(X_{v \to u}) \right) \downarrow v \end{aligned}$$

as desired.

If v is a local function node, then for each $u \in n(v)\backslash w$, $x_{\{u,v\}} = u$, and, by assumption, the message $\mu_{u\to v}$ is $f_{v\to u}(X_{v\to u}) \downarrow u$, unless u is a leaf node, in which case $f_{v\to u}(X_{v\to u}) = f_{v\to u}(u) = 1$. Now, applying the axioms defining the summary operator and Lemma 1, we write

$$\begin{aligned}
f_{w\to v}(X_{w\to v}) \downarrow w &\overset{(a)}{=} \left[v(X_{n(v)}) \prod_{u\in n(v)\backslash\{w\}} f_{v\to u}(X_{v\to u})\right] \downarrow w \\
&\overset{(b)}{=} \left[\left(v(X_{n(v)}) \prod f_{v\to u}(X_{v\to u})\right) \downarrow n(v)\right] \downarrow w \\
&\overset{(c)}{=} \left[v(X_{n(v)}) \cdot \left(\left(\prod f_{v\to u}(X_{v\to u})\right) \downarrow n(v)\right)\right] \downarrow w \\
&\overset{(d)}{=} \left[v(X_{n(v)}) \cdot \prod (f_{v\to u}(X_{v\to u}) \downarrow n(v))\right] \downarrow w \\
&\overset{(e)}{=} \left[v(X_{n(v)}) \cdot \prod (f_{v\to u}(X_{v\to u}) \downarrow u)\right] \downarrow w \\
&\overset{(f)}{=} \left[v(X_{n(v)}) \cdot \prod \mu_{u\to v}(u)\right] \downarrow w \\
&\overset{(g)}{=} \mu_{v\to w}(w).
\end{aligned}$$

In all cases, the product is over $u \in n(v) \setminus \{w\}$ as noted explicitly in equality (a), which follows from the definition of $f_{w\to v}(X_{w\to v})$. Equality (b) follows from Axiom 1, since $\{w\} \subset n(v)$, and (c) follows from Axiom 2. Since, for distinct u_1 and u_2 in $n(v) \setminus \{w\}$, $X_{v\to u_1}$ and $X_{v\to u_2}$ are disjoint, we get (d) from Axiom 3. Then (e) follows since $X_{v\to u} \cap n(v) = \{u\}$, while (f) follows from (24) by assumption, and (g) follows by definition. In other words, the message sent from v to w is the required summary. ▮

From this theorem and from Lemma 1, we obtain a proof of the correctness of the termination condition (23). We express this as a corollary.

Corollary 1 *Let F be a factor graph for a function $g(X_S)$, and suppose F is a finite tree. Let x be any variable node of F, and, for any $f \in n(x)$, let $\mu_{f\to x}(x)$ denote the message passed from f to x during the operation of the sum-product algorithm following a generalized forward/backward schedule. Then*

$$g(X_S) \downarrow x = \prod_{f\in n(x)} \mu_{f\to x}(x).$$

Proof: We write

$$g(X_S) \downarrow x = \left(\prod_{f\in n(x)} f_{x\to f}(X_{x\to f})\right) \downarrow x = \prod_{f\in n(x)} (f_{x\to f}(X_{x\to f}) \downarrow x) = \prod_{f\in n(x)} \mu_{f\to x}(x).$$

The first equality follows from the fact that $g(X_S)$ can be written as the product of the local functions contained in the disjoint subtrees obtained by removing node x from F. The second equality follows from Lemma 1 (since for distinct f_1 and f_2 in $n(x)$, we have $X_{x \to f_1} \cap X_{x \to f_2} = \{x\}$) and the third equality follows from Theorem 3. ▮

4.4.3 An Example (continued)

Let us continue now the example of Section 1.3, which corresponds to the factor graph of Fig. 1. We use an abbreviated notation, writing, e.g., ABC for the product $f_A f_B f_C$ of local functions.

Applying the GFB schedule, we pass messages as shown in Fig. 18. At each node, the sum-product rule is followed, i.e., the product of local functions is summarized for the variable associated with the given edge. Observe that the product of the messages passed in the two directions along an edge contain *all* of the factors of the global function. This observation gives us the termination rule: for each x_i, we compute the product of the messages passed in the two directions on any edge incident on x_i.

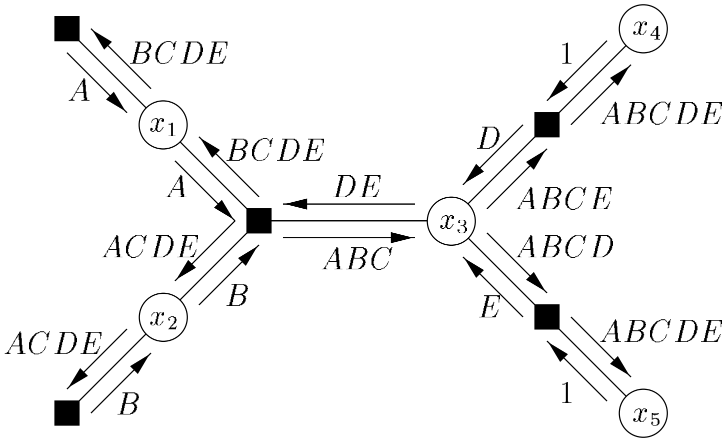


Figure 18: Messages passed during the generalized forward/backward schedule.

4.4.4 Forests

Of course, the operation of the sum-product algorithm in a tree generalizes trivially to the case when F is a "forest," i.e., a finite collection of components, each of which is a tree. Suppose, in this case, that F has K components, so that $g(X_S)$ may be written as

$$g(X_S) = \prod_{k=1}^{K} f_k(X_{C_k})$$

where C_i is an index set for the variables in the kth component with $C_k \cap C_l = \emptyset$ if $k \neq l$, and f_k is the product of local functions in the kth component. To compute $g(X_S) \downarrow x_i$ for some $i \in S$, we observe that if $i \notin C_k$ then

$$f_k(X_{C_k}) \downarrow x_i = f_k(X_{C_k}) \downarrow X_\emptyset.$$

Since by Axiom 3, the summary of the product of functions with disjoint arguments is the product of the summaries, we must simply run the sum-product algorithm separately in each component, and produce an overall summary $\rho_k = f_k(X_{C_k}) \downarrow X_\emptyset$ for each component. (This could be done by summarizing the marginal function $f_k(X_{C_k}) \downarrow x_i$ for any $i \in C_k$.) Then

$$g(X_S) \downarrow x_i = \left(f_j(X_{C_j}) \downarrow x_i\right) \cdot \prod_{\substack{k=1 \\ k \neq j}}^{K} \rho_k$$

where j is the index of the component containing x_i. Thus the sum-product algorithm can be applied to compute exact marginal functions in any finite factor graph not containing cycles.

4.5 Message Semantics under General Schedules

We have seen that when a GFB schedule is followed in a tree, each message passed by the sum-product algorithm has a clear meaning, expressed by the articulation principle. In this subsection, we consider message semantics under more general schedules.

As in Section 4.4, the behavior of the sum-product algorithm in a finite tree is easy to analyze, even with an arbitrary schedule. Let F be a factor graph that is a finite tree, let v and w be two arbitrary nodes in F, and let $\{v_1, v_2\}$, $\{v_2, v_3\}$, ... , $\{v_{L-1}, v_L\}$ (with $v_1 = v$ and $v_L = w$) denote the sequence of distinct edges in the unique path from v to w. Although nodes receive messages only from their immediate neighbors, we will say that node v *influences* node w at time t_L if there is a sequence of times $t_1 < t_2 < \cdots < t_{L-1} < t_L$ such that a pending message was sent from node v_i to v_{i+1} at time t_i. In this sequence, we say that the first message in the sequence, sent at time t_1, *reaches* w at time t_L. Note that if w has received a message from v, it has necessarily received a message from all nodes on the path from v to w. Thus, if $T^w(t)$ denotes the set of nodes which have influenced w at time t or earlier, then $T^w(t)$ induces a subtree of F.

Let v be a neighbor of w in $T^w(t)$ and let $T^w_{w\to v}(t)$ denote the component of $T_w(t)$ containing v when the edge $\{v, w\}$ is cut. By the "most recent message" sent from v to w at time t, we mean the message sent from node v to node w at time i, where i is as large as possible, but satisfies $i \leq t$. The analogy of Theorem 3 is the following.

Theorem 4 *If v and w are neighbors in $T^w(t)$ then the most recent message sent from v to w at time t in the operation of the sum-product algorithm is equal to the product of the local functions in $T^w_{v \to w}(t)$, summarized for $x_{\{v,w\}}$.*

Proof Sketch: At each node in $T^w(t)$, each processor will have followed the articulation principle in $T^w(t)$. Thus, using arguments identical to those used in the proof of Theorem 3, we obtain the desired result. The details are omitted. ▮

As already noted, in a finite tree, every nowhere idle schedule will eventually arrive at a time in which there are no nodes with messages pending. At this time, call it time t_f, every node will have received messages sent from every other node, and, in particular, the subgraph $T^w_{w \to v}(t_f)$ will be equal to $F_{w \to v}$. Theorem 4 then implies that the algorithm will arrive at the same result as the sum-product algorithm with the GFB schedule of Section 4.4. We express this result as a corollary to Theorem 4.

Corollary 2 *In a factor graph F that is a finite tree, every nowhere idle message-passing schedule for the sum-product algorithm will eventually result in a state in which no messages are pending, at which point the algorithm can be terminated to arrive at exactly the same result as obtained from a GFB schedule.*

In graphs with cycles, with one small exception (noted below), we know of no generalization of Theorems 3 and 4. The difficulty is that nodes involved in a cycle can influence themselves, at which point the clear message semantics involving the articulation principle no longer apply. Let us define as *corrupted* any node that influences itself along a path of length greater than two. (It is impossible for a node v to influence itself along a path of length two, since if v sends a message to w, the reply received from w contains no component of the message from v.) Let $F^w_{w \to v}(t)$ denote the set of nodes that have influenced node v in the latest message passed from node v to node w at time t. By definition, each node u in $F^w_{w \to v}(t)$ influences w via some chain of messages along a path from u to w (through v), and each such chain of messages originates at a definite time. Let $\tau_{u \to w \to v}(t)$ denote the largest time at which a message sent from u reaches v through w no later than time t.

Our exception is the following. Provided that every node u in $F^w_{w \to v}(t)$ is not corrupted at time $\tau_{u \to w \to v}(t)$—which necessarily implies that $F^w_{w \to v}(t)$ is a tree—the articulation principle still applies, and the latest message sent from v to w is equal to the product of the local functions in $F^w_{w \to v}(t)$, summarized for $x_{\{v,w\}}$. Thus, until "cycles close," message semantics remain clear, and are expressed by the articulation principle.

In graphs with cycles, the basic sum-product algorithm does not compute exact marginals. Although we will discuss some methods for handling these difficulties in Section 6, for some applications (e.g., decoding low-density parity-check codes and turbo codes) a successful strategy is simply to ignore the cycles, and proceed with some nowhere idle message-passing schedule, terminating the computation at some convenient point. It is difficult to analyze the behavior of iterative algorithms based on this approach, yet their excellent performance (confirmed by extensive simulation results) has led to an explosion of interest in such methods in the coding theory community.

5 Applications of the Sum-Product Algorithm

We now apply the sum-product algorithm to many of the factor graphs of Section 2. Our main results will be the derivation of a variety of well known algorithms as special cases of the sum-product algorithm.

5.1 The Forward/Backward Algorithm

The forward/backward algorithm, sometimes referred to in coding theory as the BCJR algorithm [4] or "MAP" algorithm, is an application of the sum-product algorithm to the hidden Markov model of Example 6, shown in Fig. 9(d), or to the trellises of examples Examples 3 and 4 (Figs. 6 and 7) in which certain variables are observed at the output of a memoryless channel. The local functions represent (real-valued) conditional probability distributions, and the summary operator is real addition.

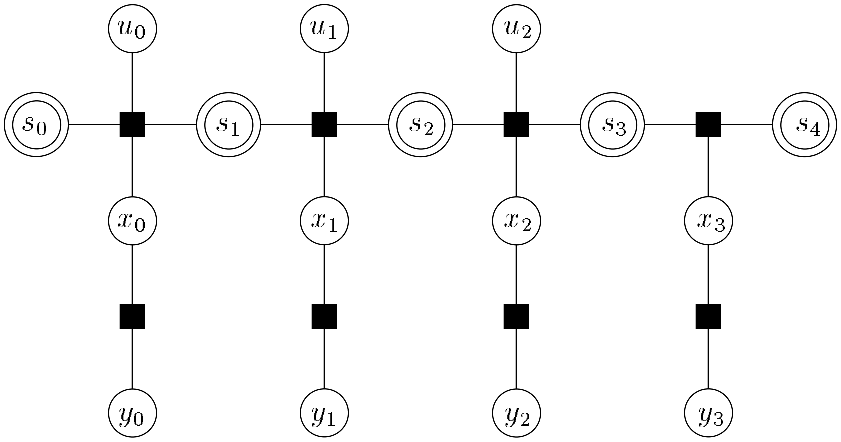


Figure 19: The factor graph on which the forward/backward algorithm operates: the s_i are state variables, the u_i are input variables, the x_i are output variables, and each y_i is the observation at the output of a memoryless channel with x_i at the input.

The factor graph of Fig. 19 represents the joint probability distribution for random vectors $\mathbf{u}$, $\mathbf{s}$, $\mathbf{x}$, and $\mathbf{y}$. Such a factor graph would arise, for example, in a terminated trellis code with time-i inputs $\mathbf{u}_i$, states $\mathbf{s}_i$, transmitted symbols $\mathbf{x}_i$ and received symbols $\mathbf{y}_i$. Given the observation of the $\mathbf{y}$, we would like to find, for each i, $p(u_i|\mathbf{y})$, the *a* *posteriori* probabilities (APPs) for the input symbols. If we take ordinary real addition as a summary operator, then, for each i,

$$p(u_i|\mathbf{y}) = p(\mathbf{u}, \mathbf{s}, \mathbf{x}|\mathbf{y}) \downarrow u_i.$$

Since the factor graph of Fig. 19 is cycle-free, these quantities can be computed using the sum-product algorithm.

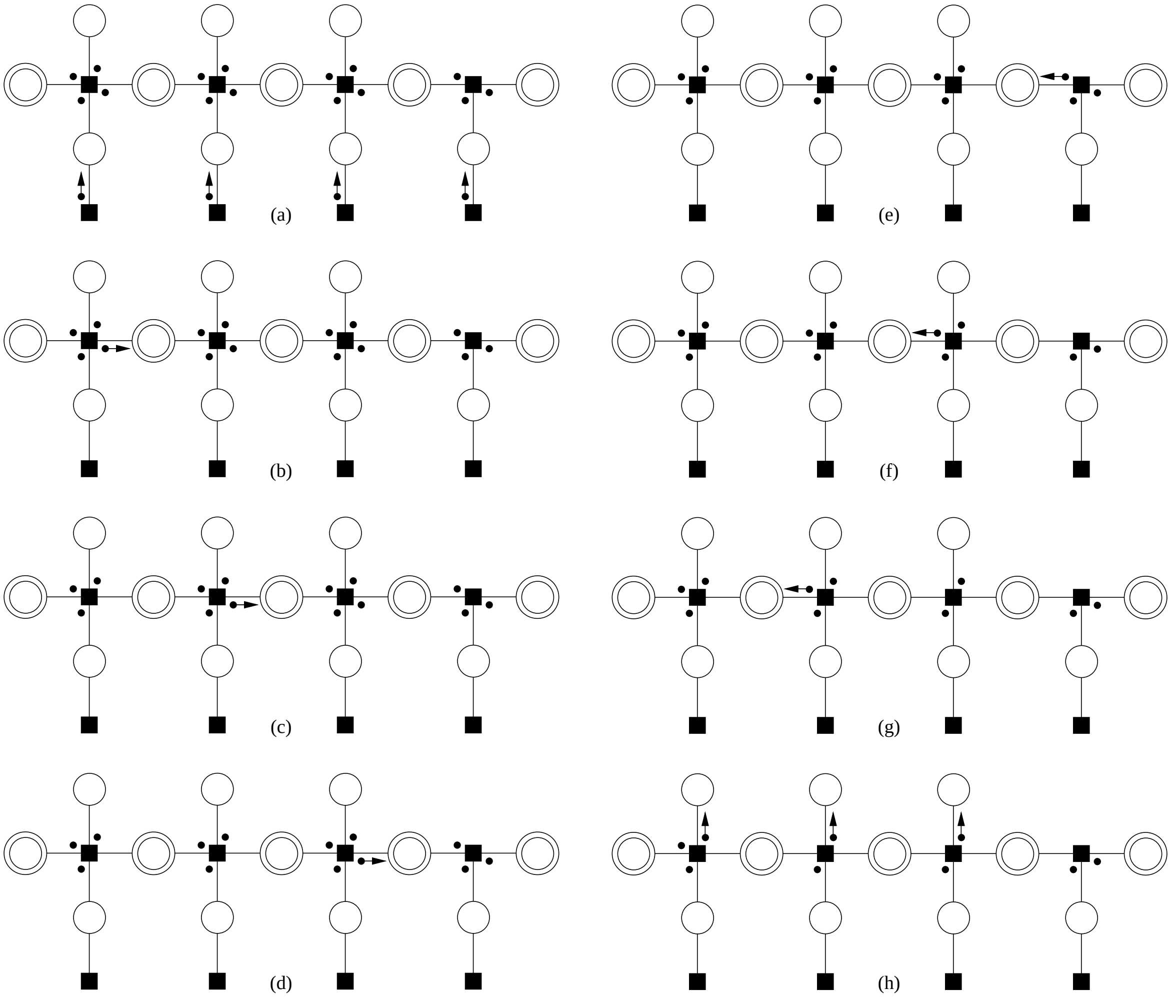


Figure 20: The typical forward backward message-passing schedule: (a) initialization; (b)–(d) the forward recursion; (e)–(g) the backward recursion; (h) termination.

Fig. 20 shows a typical schedule for the sum-product algorithm in the trellis case, for the factor graph representing the conditional probability mass function for $\mathbf{u}$, $\mathbf{s}$, and $\mathbf{x}$, given a fixed $\mathbf{y}$. Since all variable nodes have degree less than three, as described in Section B, no computation is performed at the variable nodes. Note that, except for initialization and termination steps, the main schedule involves a chain of messages passing left-to-right (or *forward*) and another chain passing right-to-left (or *backward*) in the factor graph. For this reason, the algorithm is often referred to as the forward/backward algorithm. In fact, the forward and backward message chains do not interact, so their computation could occur in parallel.

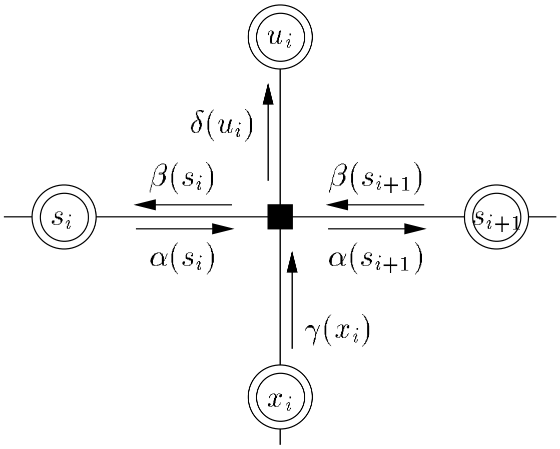


Figure 21: A detailed view of the messages passed during the operation of the forward/backward algorithm.

In the literature on the forward/backward algorithm (e.g., [4]), the messages sent from the channel input variables are referred to as 'γ's, the messages sent from state variables in the forward step are referred to as 'α's, and the messages sent from state variables in the backward step are referred to as 'β's. We refer to the conditional APP for $\mathbf{u}_i$ given $\mathbf{y}$ as $\delta(u_i)$.

Fig. 21(a) gives a detailed view of these messages for a single trellis section. The local function in this figure represents the function $[(s_i, u_i, x_i, s_{i+1}) \in T_i]$ that indicates the valid (state, input, output, next state) 4-tuples in the ith trellis section. Specializing the general update equation (20) to this case, we find:

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

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; in effect, 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

$$\begin{aligned} \alpha(s_{i+1}) &= \sum_{e \in E_i(s_{i+1})} \alpha(e)\gamma(e) \\ \beta(s_i) &= \sum_{e \in E_i(s_i)} \beta(e)\gamma(e). \end{aligned} \tag{25}$$

The basic operation in the forward and backward recursion is, therefore, one of "multiply and accumulate."

In light of Theorem 3 and the articulation principle, $\alpha(s_i)$ is 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 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 [33] for a tutorial on some of the applications of the forward/backward algorithm to applications in signal processing.

5.2 The Min-Sum Semiring and the Viterbi Algorithm

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

One way to accomplish MLSD would be to operate the sum-product algorithm in the "max-product" semiring, replacing real summation 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. Denoting by $\downarrow$ the summary operator so obtained, and denoting by $p(\mathbf{x}|\mathbf{y})$ the joint conditional probability mass function of the codeword symbols given the channel output $\mathbf{y}$, the quantity

$$p(\mathbf{x}|\mathbf{y}) \downarrow X_\emptyset \tag{26}$$

denotes the APP of the most likely sequence. Of course, we will be interested not only in determining this probability, but also in finding a valid codeword $\mathbf{x}$ that achieves this probability. Since all trellises that we consider are one-to-one, i.e., there is a unique state sequence $\mathbf{s}$ corresponding to each codeword $\mathbf{x}$, we have

$$p(\mathbf{x}|\mathbf{y}) = p(\mathbf{x}, \mathbf{s}|\mathbf{y})$$

so we can substitute the latter quantity in (26).

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 $f(\mathbf{x}, \mathbf{s}|\mathbf{y}) = -a \ln p(\mathbf{x}, \mathbf{s}|\mathbf{y}) + b$ where a and b are any convenient constants with $a > 0$, and let $\downarrow$ denote the summary operator in the min-sum semiring. We are interested not only in determining $f(\mathbf{x}, \mathbf{s}|\mathbf{y}) \downarrow X_\emptyset$, which can be achieved using the sum-product algorithm—here called the *min-sum* algorithm—whenever the factor graph is cycle free, but in determining a pair $(\mathbf{x}, \mathbf{s})$ that achieves this quantity.

Let us consider the special case of a trellis of length L (e.g., the factor graph of Fig. 19 shows $L = 4$) but ignoring the input symbols $\mathbf{u}$. Since we are interested in the overall summary $f(\mathbf{x}, \mathbf{s}|\mathbf{y}) \downarrow X_\emptyset$ we can compute $f(\mathbf{x}, \mathbf{s}|\mathbf{y}) \downarrow s_i$ for any s_i and then minimize this quantity over all values for s_i. In particular, suppose that $S_L = \{0\}$, i.e., that there is only a single terminating state. In this case, $f(\mathbf{x}, \mathbf{s}|\mathbf{y}) \downarrow s_L = f(\mathbf{x}, \mathbf{s}|\mathbf{y}) \downarrow X_\emptyset$. The former quantity can be computed by a forward recursion only, i.e., by applying steps (a)-(d) in the schedule shown in Fig. 20.

For a trellis edge $e = (s_i, x_i, s_{i+1})$ in the ith trellis section, let $\alpha(e) = \alpha(s_i)$ and $\gamma(e) = \gamma(s_i)$, where the αs and γs are the quantities (called state metrics and branch metrics, respectively) that correspond to similar quantities computed in the forward step of the forward/backward algorithm. The basic update equation corresponding to (25) then translates to

$$\alpha(s_{i+1}) = \min_{e \in E_i(s_{i+1})} (\alpha(e) + \gamma(e)), \tag{27}$$

so that the basic operation is one of "add, compare, and select" If $\gamma(e)$ is interpreted as the "cost" of traversing an edge e, this procedure computes the minimum cost of traversing from the initial state s_0 to the final state s_L, and is an example of forward dynamic programming.

Since we wish not only to compute the maximum likelihood value, but also the sequence that achieves this maximum, we need some memory of the decisions made as the αs are updated. This can be done in a variety of ways, for example by adjoining to the αs a "survivor string" m that records the sequence of output symbols on the best path from s_0 to each s_i. Initially $m(s_0) = \epsilon$, the empty string. Survivor strings strings are updated as follows: if $e^* = (s_i^*, x_i^*, s_{i+1})$ is an edge that achieves the minimum of (27), then

$$m(s_{i+1}) = m(s_i^*)|x_i^*$$

where | denotes the string concatenation operator. At termination, $m(s_L)$ is a sequence of output symbols (i.e., a codeword) corresponding to a sequence achieving the maximum likelihood value. Of course, if we wish, m could be used to store the corresponding input sequence. The unidirectional min-sum algorithm applied to a trellis and modified to have some memory of survivor sequences is usually referred to as a *Viterbi algorithm.*

5.3 Iterative Decoding of Turbo-like Codes

One of the most exciting applications of the sum-product algorithm is in the iterative decoding of near-capacity achieving codes such as turbo codes and low-density parity-check codes. Extensive simulation results (see, e.g., [7, 25, 26]) 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. Descriptions of the way in which the sum-product algorithm is applied to a variety of "compound codes" are given in [22]. In this section, we restrict ourselves to two examples: turbo codes [7] and low-density parity-check codes [14].

Turbo Codes

A "turbo code" or parallel concatenated convolutional code has the encoder structure shown in Fig. 22(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. 22(b)) are also transmitted over the channel.

A factor graph representation for a (very) short turbo code is shown in Fig. 22(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.

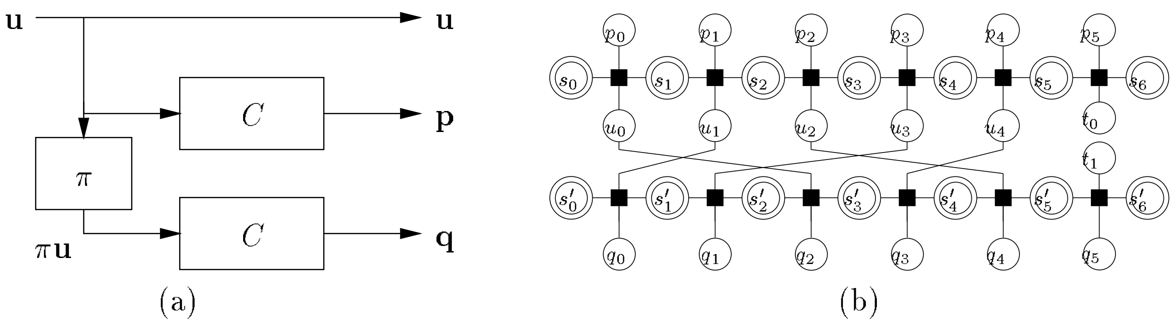

u
u
C
p
π
C
q
πu
(a)
p_0 p_1 p_2 p_3 p_4 p_5
s_0 s_1 s_2 s_3 s_4 s_5 s_6
u_0 u_1 u_2 u_3 u_4
t_0
t_1
s'_0 s'_1 s'_2 s'_3 s'_4 s'_5 s'_6
q_0 q_1 q_2 q_3 q_4 q_5
(b)

Figure 22: Turbo code: (a) encoder block diagram, (b) factor graph.

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 [22, Fig. 10].

Low-density Parity-check Codes

Low-density parity-check (LDPC) codes were introduced by Gallager [14] 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. 23 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 1.

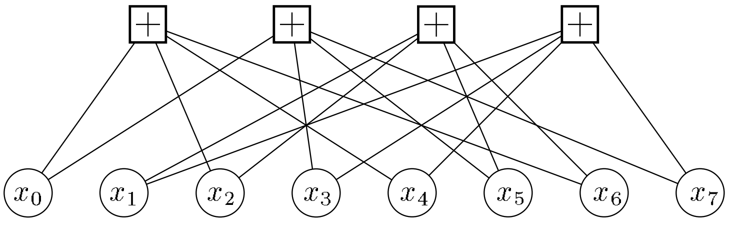


Figure 23: A factor graph for a low-density parity-check code.

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 [25, 26].

Appendix C gives some specific simplications for binary codes that are very useful in practical implementations of the min-sum and sum-product algorithms.

5.4 Belief Propagation in Bayesian Networks

Recall from Example 8 that a Bayesian network is defined in terms of a directed acyclic graph, in which each vertex represents a variable, and which represents a joint probability distribution that factors as in (12). Bayesian networks are widely used in a variety of applications in artificial intelligence and expert systems, and an extensive literature on them exists. See [31, 20] for textbook treatments.

To convert a Bayesian network into a factor graph is straightforward; we introduce a function node for each factor $p(v_i|\mathbf{a}(v_i))$ in (12) and draw edges from this node to v_i and its parents $\mathbf{a}(v_i)$. Often, we will denote the child v_i by drawing an arrow on the edge from the function node to v_i. An example conversion from a Bayesian network to a factor graph is shown in Fig. 10(c).

It turns out that Pearl's belief propagation algorithm [31] operating on a Bayesian network is equivalent to the sum-product algorithm operating on the corresponding factor graph. Equations similar to Pearl's belief updating and bottom-up/top-down propagation rules [31, pp. 182–183] can easily be derived from the general sum-product algorithm update equations (19) and (20) as follows. Again, as in the forward/backward example, local functions represent conditional probability distributions, and the summary operator is real addition.

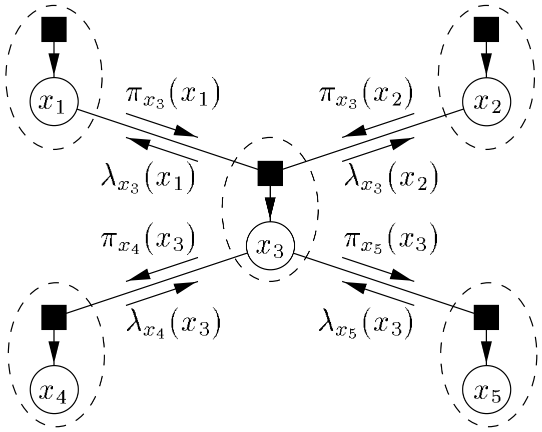


Figure 24: Messages sent in belief propagation.

In belief propagation, messages are sent between "variable nodes," corresponding to the dashed ellipses for the particular Bayesian network shown in Fig. 24. 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 [31], 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. 24 for the specific Bayesian network of Fig. 10(c).

Consider the central variable, x_3 in Fig. 24. Clearly the message sent upwards by the sum-product algorithm to the local function f contained in the ellipse is, from (19), 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 (20), 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 $a \in \mathbf{a}(x)$,

$$\lambda_x(a) = \left(\prod_{d\in\mathbf{d}(x)}\lambda_d(x)f(x|\mathbf{a}(x))\prod_{p\in\mathbf{a}(x)\setminus\{a\}}\pi_x(p)\right)\downarrow a. \tag{28}$$

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

$$\pi_d(x) = \prod_{c\in\mathbf{d}(x)\setminus\{d\}}\lambda_c(x)\left(f(x|\mathbf{a}(x))\prod_{a\in\mathbf{a}(x)}\pi_x(a)\right)\downarrow x. \tag{29}$$

The termination condition for cycle-free graphs, called the "belief update" equation in [31], is given by the product of the messages received by x in the factor graph:

$$BEL(x) = \prod_{d\in\mathbf{d}(x)}\lambda_d(x)\left(f(x|\mathbf{a}(x))\prod_{a\in\mathbf{a}(x)}\pi_x(a)\right)\downarrow x. \tag{30}$$

Pearl also introduces a scale factor in (29) and (30) so that the resulting messages properly represent probability mass functions. The relative complexity of (28)–(30) compared with the simplicity of the sum-product update rule given in Section 4 provides a strong pedagogical incentive for the introduction of factor graphs.

Pearl also presents an algorithm called "belief revision" in [31]; in our terms, belief revision is the "max-product" version of the sum-product algorithm, applied to the factor graph corresponding to a Bayesian network. The details of this straightforward extension are omitted.

5.5 Kalman Filtering

In this section, we derive the Kalman filter [3] as the optimal predictor given by the sum-product algorithm in a factor graph for a time-varying discrete-time linear dynamical system (*cf.*, (8)). The input to the system consists of a sequence of unknown K-dimensional real-valued input column vectors u_j, $j = 1, 2, \ldots$. A sequence of hidden N-dimensional real-valued state vectors x_j, $j = 1, 2, \ldots$ are meant to represent the internal dynamics of the system. Input vector u_j and state vector x_j combine linearly to produce the next state vector:

$$x_{j+1} = A_j x_j + B_j u_j. \tag{31}$$

A_j is the $N \times N$ state transition matrix at time j and B_j is an $N \times K$ matrix that maps the input into the state space. The input vectors may also be interpreted as state transition noise vectors.

The output of the system is a sequence of M-dimensional real-valued vectors y_j, $j = 1, 2, \ldots$. The output y_j is a linear function of the state vector x_j plus a linear function of an L-dimensional noise vector, w_j:

$$y_j = C_j x_j + D_j w_j. \tag{32}$$

C_j is the $M \times N$ output matrix at time j and D_j is an $N \times L$ matrix that maps the noise process into the output space. Notice that even though the output space may be high-dimensional, the output noise may be low-dimensional, $L \ll N$. In this paper, we consider the case where the A, B, C and D matrices are given.

Assume that the input vectors and the output noise vectors are all independent and Gaussian with zero mean. Let the $K \times K$ covariance matrix of u_j be ϕ_j:

$$\phi_j = \mathrm{COV}(u_j) = \mathrm{E}\big[\big(u_j - \mathrm{E}[u_j]\big)\big(u_j - \mathrm{E}[u_j]\big)'\big] = \mathrm{E}[u_j u_j'],$$

where "$'$" indicates vector transpose. Let the $L \times L$ covariance matrix of w_j be ψ_j. Then, we can define the following conditional covariance matrices:

$$\begin{aligned} \beta_j &\triangleq \mathrm{COV}(x_{j+1}|x_j) = \mathrm{E}[(B_j u_j)(B_j u_j)'] = B_j \phi_j B_j', \\ \delta_j &\triangleq \mathrm{COV}(y_j|x_j) = \mathrm{E}[(D_j w_j)(D_j w_j)'] = D_j \psi_j D_j', \end{aligned} \tag{33}$$

where $\mathrm{E}[x_{j+1}|x_j] = A_j x_j$ and $\mathrm{E}[y_j|x_j] = C_j x_j$ were used to simplify the expressions. The state sequence is initialized by setting $x_0 = 0$, so that $x_1 = B_0 u_0$, a Gaussian vector with zero mean and covariance β_0. Since linear combinations of jointly Gaussian random variables are Gaussian, $\{x_j\}$ and $\{y_j\}$ are jointly Gaussian.

This signal structure can be written in terms of conditional probability densities. Letting

$$\mathcal{N}(x, m, \Sigma) \triangleq \frac{1}{|2\pi\Sigma|^{1/2}} \exp[-\frac{1}{2}(x - m)'\Sigma^{-1}(x - m)]$$

be the normal distribution for x with mean m and covariance matrix Σ, we have

$$\begin{aligned} p(x_{j+1}|x_j) &= \mathcal{N}(x_{j+1}, A_j x_j, \beta_j), \\ p(y_j|x_j) &= \mathcal{N}(y_j, C_j x_j, \delta_j). \end{aligned} \tag{34}$$

From the description of the linear dynamical system given above, it is clear that given x_{j-1}, x_j is independent of $x_1, \ldots, x_{j-2}$ and $y_1, \ldots, y_{j-1}$:

$$p(x_j|x_0, \ldots, x_{j-1}, y_1, \ldots, y_{j-1}) = p(x_j|x_{j-1}).$$

Also, given x_j, y_j is independent of $x_1, \ldots, x_{j-1}$ and $y_1, \ldots, y_{j-1}$:

$$p(y_j|x_0, \ldots, x_j, y_1, \ldots, y_{j-1}) = p(y_j|x_j).$$

Using the chain rule of probability, it follows that the joint density for observations up to time $t-1$ and states up to time t can be written

$$\begin{aligned} &p(x_1, \ldots, x_t, y_1, \ldots, y_{t-1}) \\ &\quad = p(x_t|x_0, \ldots, x_{t-1}, y_1, \ldots, y_{t-1}) \\ &\qquad \cdot \prod_{j=1}^{t-1} p(x_j|x_0, \ldots, x_{j-1}, y_1, \ldots, y_{j-1}) p(y_j|x_0, \ldots, x_j, y_1, \ldots, y_{j-1}) \\ &\quad = p(x_t|x_{t-1}) \prod_{j=1}^{t-1} p(x_j|x_{j-1}) p(y_j|x_j). \end{aligned}$$

The factor graph for this global probability density function is shown in Fig. 25. Note that since x_0 is fixed to zero, $p(x_1|x_0) = p(x_1)$ from (34). A factor graph for the cross-section of this function at fixed $y_1, \ldots, y_{t-1}$ is obtained by eliminating the y nodes from Fig. 25 and interpreting $p(y_j|x_j)$ as a likelihood function of x_j with parameter y_j.

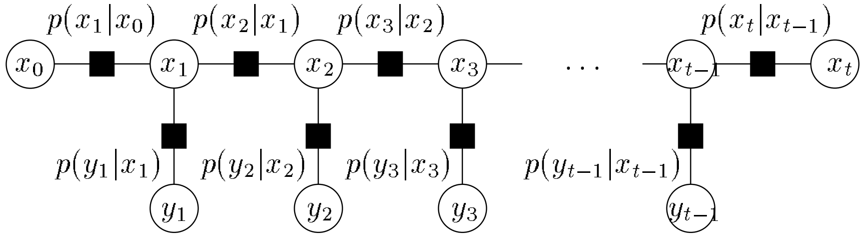


Figure 25: A factor graph for the linear dynamical system model (31) and (32).

We consider the standard state observer problem, *i.e.*, the problem of estimating the next state x_t from the past observations $y_1, \ldots, y_{t-1}$. The minimum mean squared error prediction for x_t is given by

$$E[x_t|y_1, \ldots, y_{t-1}],$$

the conditional expected value of x_t given the observed outputs. This conditional expectation can be obtained from the conditional density $p(x_t|y_1, \ldots, y_{t-1})$. In fact, we really need only compute a function of x_t that is *proportional* to $p(x_t|y_1, \ldots, y_{t-1})$, assuming we can later normalize the function. In this case, since all random variables are jointly Gaussian, all normalizing factors can be computed from the covariance matrices and hence do not need to be computed explicitly.

Since the factor graph is cycle-free, we can take integration as the summary operator (as described in Section 3.3) and directly apply the sum-product algorithm to compute a function proportional to

$$p(x_t|y_1, \ldots, y_{t-1}) = p(x_1, \ldots, x_t|y_1, \ldots, y_{t-1}) \downarrow x_t.$$

We do not have any observations for $k \geq t$, so we need only propagate messages forward along the chain.

Denote by f_j the function node in the factor graph corresponding to $p(x_j|x_{j-1})$ and by g_j the function node corresponding to $p(y_j|x_j)$. Since x_0 is fixed to zero, we take the first variable-to-function message $\mu_{x_0 \to f_1}(x_0)$ to be a delta-Dirac function at 0: $\mu_{x_0 \to f_1}(x_0) = \delta(x_0)$. From (20), it is evident that the first function-to-variable message $\mu_{f_1 \to x_1}(x_1)$ is thus

$$\begin{aligned}\mu_{f_1 \to x_1}(x_1) &= \int_{x_0} p(x_1|x_0)\mu_{x_0 \to f_1}(x_0)dx_0 \\ &= \int_{x_0} \mathcal{N}(x_1, A_0x_0, \beta_0)\delta(x_0)dx_0 = \mathcal{N}(x_1, 0, \beta_0).\end{aligned}$$

We now show how to compute the function-to-variable message $\mu_{f_{j+1} \to x_{j+1}}(x_{j+1})$ at time $j+1$ from the function-to-variable message $\mu_{f_j \to x_j}(x_j)$ at time j. First, note from (20) that the messages sent from the likelihood function nodes to the state nodes are simply

$$\mu_{g_j \to x_j}(x_j) = g_j(x_j) = p(y_j|x_j) = \mathcal{N}(y_j, C_jx_j, \delta_j).$$

From (19), the message $\mu_{x_j \to f_{j+1}}(x_j)$ sent from x_j to f_{j+1} is given by

$$\mu_{x_j \to f_{j+1}}(x_j) = \mu_{f_j \to x_j}(x_j)\mu_{g_j \to x_j}(x_j) = \mu_{f_j \to x_j}(x_j)\mathcal{N}(y_j, C_jx_j, \delta_j).$$

From (20), the message $\mu_{f_{j+1} \to x_{j+1}}(x_{j+1})$ sent from f_{j+1} to x_{j+1} is given by

$$\begin{aligned}\mu_{f_{j+1} \to x_{j+1}}(x_{j+1}) &= \int_{x_j} p(x_{j+1}|x_j)\mu_{x_j \to f_{j+1}}(x_j)dx_j \\ &= \int_{x_j} \mathcal{N}(x_{j+1}, A_jx_j, \beta_j)\mathcal{N}(y_j, C_jx_j, \delta_j)\mu_{f_j \to x_j}(x_j)dx_j. \qquad (35)\end{aligned}$$

Since the initial function-to-variable message $\mu_{f_1 \to x_1}(x_1)$ is a Gaussian and a convolution of Gaussians gives a scaled Gaussian, all further function-to-variable messages will be scaled Gaussians. Also, since the state variable at time t in the factor graph has just a single edge (see Fig. 25), from (23) we have

$$p(x_t | y_1, \ldots, y_{t-1}) \propto \mu_{f_t \to x_t}(x_t).$$

So, the predicted mean and covariance of the next state given past observations are exactly equal to the mean and covariance of the function-to-variable message at time t. These two observations mean that we really only need keep track of the means and covariances of the messages — scale factors can be ignored.

Let $\hat{x}_j$ and Σ_j be the mean and covariance for the message $\mu_{f_j \to x_j}(x_j)$:

$$\mu_{f_j \to x_j}(x_j) \propto \mathcal{N}(x_j, \hat{x}_j, \Sigma_j). \tag{36}$$

From (35) and (36), we have

$$\mu_{f_{j+1} \to x_{j+1}}(x_{j+1}) \propto \int_{x_j} \mathcal{N}(x_{j+1}, A_j x_j, \beta_j) \mathcal{N}(y_j, C_j x_j, \delta_j) \mathcal{N}(x_j, \hat{x}_j, \Sigma_j) dx_j.$$

Clearly the solution to this integral can be written in closed form, since it is simply a convolution of Gaussians. The first trick to simplifying the message is to rearrange terms in the exponents of the Gaussians in the above product to complete the square for x_j. This will produce a normal distribution over x_j, which will integrate to unity, leaving a function of only x_{j+1}. Since we are only interested in finding the mean and covariance of this message, we can ignore any normalization terms that are introduced by the rearrangement. The second trick to simplifying the message is to complete the square for x_{j+1} so that the mean and covariance of the message become evident. These two tricks produce the following message:

$$\begin{aligned}
&\mu_{f_{j+1} \to x_{j+1}}(x_{j+1}) \propto \\
&\qquad \int_{x_j} \mathcal{N}\big(x_j, [\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j]^{-1} [\hat{x}_j' \Sigma_j^{-1} + x_{j+1} \beta_j^{-1} A_j + y_j' \delta_j^{-1} C_j]', \\
&\qquad\qquad [\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j]^{-1}\big) \\
&\qquad \mathcal{N}\big(x_{j+1}, [\beta_j^{-1} - \beta_j^{-1} A_j \{\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j\}^{-1} A_j' \beta_j^{-1}]^{-1} \beta_j^{-1} A_j \\
&\qquad\qquad \cdot [\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j]^{-1} [\Sigma_j^{-1} \hat{x}_j + C_j' \delta_j^{-1} y_j], \\
&\qquad\qquad [\beta_j^{-1} - \beta_j^{-1} A_j \{\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j\}^{-1} A_j' \beta_j^{-1}]^{-1}\big) \;\; dx_j \\
&\qquad = \mathcal{N}\big(x_{j+1}, [\beta_j^{-1} - \beta_j^{-1} A_j \{\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j\}^{-1} A_j' \beta_j^{-1}]^{-1} \beta_j^{-1} A_j \\
&\qquad\qquad \cdot [\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j]^{-1} [\Sigma_j^{-1} \hat{x}_j + C_j' \delta_j^{-1} y_j], \\
&\qquad\qquad [\beta_j^{-1} - \beta_j^{-1} A_j \{\Sigma_j^{-1} + A_j' \beta_j^{-1} A_j + C_j' \delta_j^{-1} C_j\}^{-1} A_j' \beta_j^{-1}]^{-1}\big).
\end{aligned}$$

The mean and covariance for this message can be simplified by applying the following matrix inversion lemma:

$$(\Lambda^{-1} + V\Omega^{-1}V')^{-1} = \Lambda - \Lambda V(\Omega + V'\Lambda V)^{-1}V'\Lambda,$$

for $(\Omega + V'\Lambda V)^{-1}$ nonsingular. Neophytes may be interested in this exercise, but if not, rest assured that we and many others have done it. The simplified form of the message is

$$\begin{aligned}\mu_{f_{j+1}\to x_{j+1}}(x_{j+1}) \propto \mathcal{N}\big(x_{j+1}, &[A_j - A_j\Sigma_jC_j'(\delta_j + C_j\Sigma_jC_j')^{-1}C_j](\hat{x}_j + \Sigma_jC_j'\delta_j^{-1}y_j),\\ &\beta_j + A_j[\Sigma_j - \Sigma_jC_j'(\delta_j + C_j\Sigma_jC_j')^{-1}C_j\Sigma_j]A_j'\big).\end{aligned}$$

Inserting the definitions for β_j and δ_j given in (33), we find that the mean and covariance of the message $\mu_{f_{j+1}\to x_{j+1}}(x_{j+1})$ can be expressed

$$\hat{x}_{j+1} = A_j\hat{x}_j + K_j(y_j - C_j\hat{x}_j),$$

and

$$\Sigma_{j+1} = B_j\phi_jB_j' + A_j[\Sigma_j - \Sigma_jC_j'(D_j\psi_jD_j' + C_j\Sigma_jC_j')^{-1}C_j\Sigma_j]A_j',$$

where

$$K_j = A_j\Sigma_jC_j'(D_j\psi_jD_j' + C_j\Sigma_jC_j')^{-1}.$$

These updates are exactly equal to the updates used by Kalman filtering [3]. In particular, K_j is called the *filter gain* and $y_j - C_j\hat{x}_j$ is called the *innovation*, and $\hat{x}_t$ and Σ_t are the estimated mean and covariance of x_t given $y_1, \ldots, y_{t-1}$.

6 Factor Graph Transformations and Coping with Cycles

In this section we describe a number of straightforward transformations that may be applied to a factor graph without changing its meaning. 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 messages that must be sent during the operation of the sum-product algorithm. Nevertheless, such transformations can be quite useful, and we apply them to derive a fast Fourier transform algorithm from the factor graph representing the DFT kernel. In [20, 24], similar general procedures are described for transforming a graphical probability model into cycle-free form.

6.1 Grouping Like Nodes, Multiplying by Unity

It is always possible to group two nodes of like type—i.e., both variable nodes or both function nodes—without changing the global function being represented by a factor graph. If v and w are the two nodes being grouped, 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 over alphabets A_v and A_w respectively, by the "pairing of v and w" we mean a new variable (v, w) over the alphabet $A_v \times A_w$. Any function f that had, say, v as an argument in the original graph must be converted into an equivalent function f' that has (v, w) as an argument. This is accomplished simply by applying the projection operator P_v that maps (v, w) to v and writing $f'((v, w), \ldots) = f(P_v(v, w), \ldots) = f(v, \ldots)$. Note that f' still has essentially the same set of arguments as f; if w is not an argument of f, then f' does not depend on w. Thus, grouping variables does not increase 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 $X_{n(v)}$ and $X_{n(w)}$ are the arguments of v and w, then $X_{n(v)\cup n(w)}$ is the set of arguments of the product. Note that grouping functions does not increase the complexity of the variables.

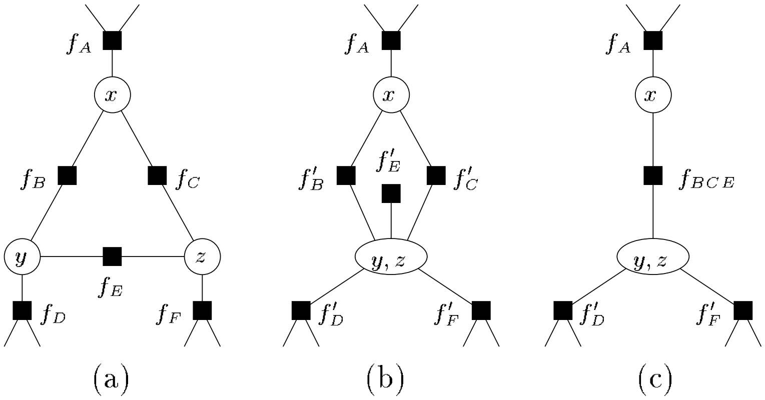


Figure 26: Grouping transformations: (a) original factor graph fragment, (b) variable nodes y and z grouped together, (c) function nodes f_B, f_C and f_E grouped together.

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. 26(a) and connecting the neighbors of both nodes to the new grouped node, we obtain the factor graph fragment shown in Fig. 26(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 for 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 for the old factor graph.

In Fig. 26(b), there is still one cycle; however, it can easily be removed by grouping function nodes. In Fig. 26(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 $|A_y||A_z|$.

We will also allow the introduction of arbitrary factors of unity. Essentially, if convenient, for any set X_B of variables, we can multiply the global function g by a factor of unity, $f(X_B) = 1$, and introduce the corresponding function node and edges in the factor graph for g.

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 $x_{\{v,w\}}$ 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. 27(a) shows a factor graph, and Fig. 27(b) shows an equivalent factor graph in which x_1 has been stretched to all variable nodes.

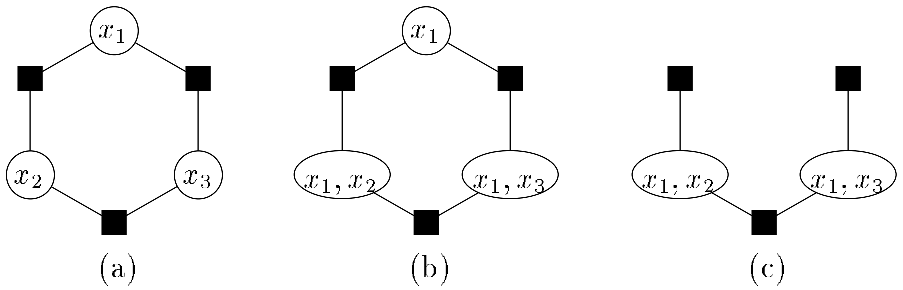


Figure 27: Stretching transformation: (a) original factor graph, (b) node x_1 is stretched to x_2 and x_3, (c) the node representing x_1 alone is now redundant and can be removed.

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. 9(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. 27(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. 27(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. 27(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 increase in the complexity of the variable alphabets.

In the next subsection we present a method for coping with the cycles in a factor graph based on forming a spanning tree for the graph. Each spanning tree is essentially obtained by stretching variables and deleting redundant edges.

6.3 Spanning Trees

Recall that a spanning tree T for a connected graph G is a connected, cycle-free subgraph of G having the same vertex set as G. In general, if G is not itself a tree, there are many spanning trees for G, each of which is obtained from G by deleting some of the edges of G. (If G has more than one component, then one could construct a tree spanning each component, resulting in a "spanning forest" for G. However, we will not consider this obvious extension further.)

Let $F(S, Q)$ be a factor graph of one component, and for every variable node x of F, let $n(x)$ denote the set of neighbors of x in F, i.e., the set of function nodes having x as an argument. Now, let T be a tree spanning F. 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.

Theorem 5 *Every edge of F' not in T is redundant and all such edges can be deleted from F'.*

Proof: Let e be 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 theorem 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." To paraphrase the California winemakers Gallo, "we summarize no variable before its time."

6.4 An 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 translate the approach of Aji and McEliece to the language of factor graphs.

A factor graph for the DFT kernel was given in Section 2, Example 11. We observed in (15) that the DFT W_k of the sequence w_n could be obtained as a marginal function.

The factor graph in Fig. 28(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. 28(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. 28(b), essentially by defining

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

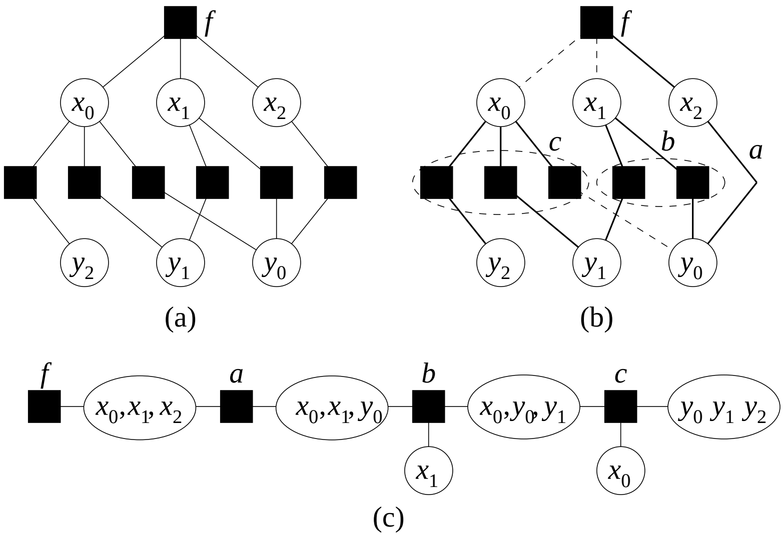


Figure 28: The discrete Fourier transform kernel: (a) factor graph; (b) a particular spanning tree; (c) spanning tree after clustering and stretching transformation.

we arrive at the spanning tree shown in Fig. 28(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.

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 defined the sum-product algorithm in terms of two basic function operations: function product and function summary. The function product was defined in general terms involving an arbitrary semigroup with unity. Likewise the summary operation was defined in general terms as an operator that converts the projection of a function on a subset of the coordinates of its domain into a function. An important contribution of this paper is a clear set of axioms that such a summary operator should possess. The flexibility obtained from defining these concepts in general terms is one reason for the broad applicability of the sum-product algorithm.

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.

Another major motivation in writing this paper was pedagogical: we feel that many of the topics presented in this paper can and should be taught in a unified manner using the framework presented in this paper. After learning to apply a single simple computational procedure, the student immediately has access to a wide variety of algorithms in different application areas.

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 Proof of Theorem 1

Assume that $p(X_S)$ factors as in (2), i.e., $p(X_S) = \prod_{E \in Q} f_E(X_E)$, for some collection Q of subsets of the index set S, where $f_E(X_E)$ is a non-negative real-valued function for all $E \in Q$. Let $F(S, Q)$ be the corresponding factor graph. We must show that F_S^2 satisfies the local Markov property (10). Denote by $n(x_i)$ the set of neighbors of x_i in F^2.

Adopting the index/subset view of a factor graph, we consider an arbitrary but fixed node $i \in S$. Let $Q_i \subset Q$ be the set of neighbors of i in F, and let $R_i = Q \setminus Q_i$ be the subset nodes that are *not* neighbors of i in F. For every $E \in R_i$, f_E does not have x_i as an argument. We then write

$$\begin{aligned}
p(x_i | X_S \setminus \{x_i\}) &= \frac{p(X_S)}{\sum_{x_i} p(X_S)} \\
&= \frac{\prod_{E \in Q} f_E(X_E)}{\sum_{x_i} \prod_{E \in Q} f_E(X_E)} \\
&= \frac{\prod_{E \in R_i} f_E(X_E) \prod_{E \in Q_i} f_E(X_E)}{\prod_{E \in R_i} f_E(X_E) \sum_{x_i} \prod_{E \in Q_i} f_E(X_E)} \\
&= p(x_i | n(x_i)).
\end{aligned}$$

The latter equality is shown as follows. Let J be an index set for the variables other than x_i and those in $n(x_i)$, and observe that for every $E \in Q_i$, f_E has no variable of X_J as an argument. Then

$$\begin{aligned}
p(x_i | n(x_i)) &= \frac{\sum_{X_J} p(X_S)}{\sum_{x_i} \sum_{X_J} p(X_S)} \\
&= \frac{\prod_{E \in Q_i} f_E(X_E) \sum_{X_J} \prod_{E \in R_i} f_E(X_E)}{\sum_{x_i} \prod_{E \in Q_i} f_E(X_E) \sum_{X_J} \prod_{E \in R_i} f_E(X_E)} \\
&= \frac{\prod_{E \in Q_i} f_E(X_E)}{\sum_{x_i} \prod_{E \in Q_i} f_E(X_E)}.
\end{aligned}$$

Alternatively, we can argue as follows. Observe that the variables in X_E form a *clique* in F_S^2, and that $f_E(X_E)$ can be taken as a Gibbs potential function over this clique. By assigning a unit potential function to all cliques of F_S^2 that do not correspond to some $E \in Q$, we obtain a collection of Gibbs potential functions over all of the cliques of F_S^2. It is well known (see, e.g., [21]) that any such collection of non-negative Gibbs potential functions is linearly proportional to a probability distribution that satisfies the local Markov property (10) and hence defines a Markov random field. ▮

B Complexity of the Sum-Product Algorithm in a Finite Tree

The complexity of the sum-product algorithm is difficult to analyze in general, depending as it does on implementation-dependent details such as the manner in which local functions are represented, the mapping from messages to functions and back, the available storage, the complexity of multiplying elements in R, and of evaluating a summary, etc. One way to measure complexity would be simply to count messages, in which case the complexity of the sum-product algorithm would be $2|E|$ in a tree with $|E|$ edges. However, this does not account for the (in general, highly variable) complexity needed to compute the values of the messages. In this section we give a simplistic complexity analysis for the sum-product algorithm operating in a finite tree.

We will assume that all variable alphabets are finite, denoting the size of the alphabet A_i for x_i by $|A_i|$. We will assume that a local function $f(X_E)$ is represented by a table of $|f_E| = \prod_{i \in E} |A_i|$ values, and that function evaluation is performed by table lookup. Furthermore, we will assume that all messages passed by the sum-product algorithm describe the corresponding function by an ordered list of function values. We will assume that the summary operator is defined in terms of a binary addition operation in a semiring $R(\cdot, +)$. We will assume no capability to store partial results at a node, other than the partial results involved in the computation of a particular message. At a node v, we denote by $\sigma(v)$, $\chi(v)$, and $\lambda(v)$ the number of additions, multiplications and local function evaluations (table-lookups) required at v to generate all messages needed over the course of operation of the sum-product algorithm.

At a variable node x_i, the message $\mu_{x_i \to f}$ sent to a neighbor f is—from (19)—simply the component-wise product of messages received on all other edges. In particular, no additions or local function evaluations need to be performed, hence

$$\sigma(x_i) = \lambda(x_i) = 0.$$

For every outgoing message, there are $(\partial(x_i) - 1)$ incoming messages, each of length $|A_i|$. If $\partial(x_i) > 2$, these can be multiplied together using $\partial(x_i) - 2$ multiplies per component, or $|A_i|(\partial(x_i) - 2)$ multiplies in total. (Otherwise, if $\partial(x_i) \leq 2$, no work needs to be done at the variable node; the outgoing message is equal to the incoming message.) Over the course of operation of the sum-product algorithm, this operation is done for *each* neighbor, and since we assume no storage of partial results, the number of multiplies performed to compute messages sent from x_i is

$$|A_i|(\partial(x_i))(\partial(x_i) - 2)[\partial(x_i) > 2],$$

where we have used Iverson's convention, described in Section 1, to express the condition on the degree of x_i. The marginal function for x_i can be computed by multiplying the messages sent and received on any given edge at x_i; this can be done using another $|A_i|$ multiplies, so the total number of multiplies at x_i is

$$\chi(x_i) = |A_i|(\partial(x_i))(\partial(x_i) - 2)[\partial(x_i) > 2] + |A_i| = \mathcal{O}\left(|A_i|\partial^2(x_i)\right).$$

At a function node f_E, we must perform multiplications, additions, and local function evaluations according to (20). For the message $\mu_{f_E \to x_i}(x_i)$ we must first compute a table of values representing the product $f(X_E)\prod_{x_j \in X_E \setminus \{x_i\}} \mu_{x_j \to f_E}(x_j)$, which can be computed using no more than $|f(X_E)|\partial(f_E)$ multiplies, and $|f(X_E)|$ local function evaluations. (Since messages received from leaf variable nodes are trivial, the actual number of computations may be less than this value.) Since this must be done for each neighbor,

$$\chi(f_E) \leq \partial^2(f_E)|f_E| = \partial^2(f_E)\prod_{i \in E}|A_i|$$

and

$$\lambda(f_E) = \partial(f_E)|f_E| = \partial(f_E)\prod_{i \in E}|A_i|.$$

We must then summarize the multiple function values for each particular value that x_i can take on; this requires $|f_E|/|A_i| - 1$ additions per component, or $|f_E| - |A_i|$ additions in total. Since this must be repeated for each neighbor, we get

$$\sigma(f_E) = \partial(f_E)|f_E| - \sum_{i \in E}|A_i| = \partial(f_E)\prod_{i \in E}|A_i| - \sum_{i \in E}|A_i| = \mathcal{O}\left(\partial(f_E)\prod_{i \in E}|A_i|\right)$$

addition operations in total.

If we define the *alphabet complexity* $\alpha(v)$ a node v as $|A_i|$ if $v = x_i$ and $\prod_{i \in E}|A_i|$ if $v = f_E$, then the number of multiplications performed at v scales as

$$\chi(v) = \mathcal{O}\left(\alpha(v)\partial^2(v)\right)$$

and, when v is a function node,

$$\sigma(v) = \lambda(v) = \mathcal{O}(\alpha(v)\partial(v)).$$

C Code-Specific Simplifications

For particular decoding applications, the generic updating rules (19) and (20) can often be substantially simplified. We treat here only the important case where all variables are binary and all functions (except single-variable functions) are parity checks (as in Fig. 8(b) and in Fig. 23). This includes, in particular, low-density parity check codes [14] of the previous subsection. We give the corresponding simplifications for both the sum-product and the min-sum versions of the algorithm, both of which were known long ago [14]. We begin with the latter.

Min-Sum Updating

The min-sum update rule corresponding to (19) for the message $\mu_{b\to J}$ sent from a bit b to a check J is

$$\begin{bmatrix} \mu_{b\to J}(0) \\ \mu_{b\to J}(1) \end{bmatrix} = c + \begin{bmatrix} \sum_i \mu_{i\to b}(0) \\ \sum_i \mu_{i\to b}(1) \end{bmatrix}, \tag{37}$$

where $\mu_{i\to b}$, $i = 1, 2, \ldots$ are the inputs into b from all *other* parity checks and where c is an arbitrary constant.

The update rule corresponding to (20) for the message $\mu_{J\to b}$ sent from a parity check J to a bit b is as follows. For the sake of clarity, we assume that J checks two other bits besides b. Then the rule is

$$\begin{bmatrix} \mu_{J\to b}(0) \\ \mu_{J\to b}(1) \end{bmatrix} = c + \begin{bmatrix} \min\{\mu_{1\to J}(0) + \mu_{2\to J}(0), \mu_{1\to J}(1) + \mu_{2\to J}(1)\} \\ \min\{\mu_{1\to J}(0) + \mu_{2\to J}(1), \mu_{1\to J}(1) + \mu_{2\to J}(0)\} \end{bmatrix}. \tag{38}$$

Due to the arbitrary additive constant c, every message vector μ can be collapsed to a single real number

$$\lambda \triangleq \mu(1) - \mu(0). \tag{39}$$

The sign of this number can be viewed as a decision about the value of the corresponding bit (positive for 1 and negative for 0), while the magnitude of this number can be viewed as an estimate of the reliability of this decision. The bit-to-check update rule (37) then becomes

$$\lambda_{b\to J} = \sum_i \lambda_{i\to b}. \tag{40}$$

The check-to-bit rule (38) becomes

$$\lambda_{J\to b} = \left(\prod_i \operatorname{sign}(\lambda_{i\to J}) \right) \min_i\{|\lambda_{i\to J}|\}, \tag{41}$$

where we have directly stated the general case for an arbitrary number of checked bits. The first factor in (41), which is really an exclusive-OR operation, follows from noting that the least-cost configuration is obtained by making individual decision about the other bits and then setting b as needed. The second factor follows from noting that the cheapest alternative with a different value for b is obtained by flipping the least reliable other bit.

Sum-Product Updating

The sum-product update rule (19) for the function $\mu_{b\to J}$ from a bit b to parity check J is

$$\begin{bmatrix} \mu_{b\to J}(0) \\ \mu_{b\to J}(1) \end{bmatrix} = \gamma \begin{bmatrix} \prod_i \mu_{i\to b}(0) \\ \prod_i \mu_{i\to b}(1) \end{bmatrix}, \tag{42}$$

where $\mu_{i\to b}$, $i = 1, 2, \ldots$ are the inputs into b from all *other* parity checks and where γ is an arbitrary nonzero scale factor.

The update rule (20) for the function $\mu_{J\to b}$ from a parity check J to a bit b is as follows. For simplicity we assume that J checks two other bits besides b:

$$\begin{bmatrix} \mu_{J\to b}(0) \\ \mu_{J\to b}(1) \end{bmatrix} = \gamma \begin{bmatrix} \mu_{1\to J}(0)\mu_{2\to J}(0) + \mu_{1\to J}(1)\mu_{2\to J}(1) \\ \mu_{1\to J}(0)\mu_{2\to J}(1) + \mu_{1\to J}(1)\mu_{2\to J}(0) \end{bmatrix}. \tag{43}$$

Because of the (arbitrary, but nonzero) scale factor γ, every function μ can be collapsed into a single real number

$$\Lambda \triangleq \mu(0)/\mu(1). \tag{44}$$

The bit-to-check update rule (42) becomes

$$\Lambda_{b\to J} = \prod_i \Lambda_{i\to b}. \tag{45}$$

So far, the analogy between the min-sum rules and the sum-product rules is perfect. What is missing is the sum-product analog to (41), i.e., the formulation of the check-to-bit update rule (43) in terms of Λ. Towards that end, we first observe that the operation (43) on the normalized difference

$$\Delta \triangleq \frac{\mu(0) - \mu(1)}{\mu(0) + \mu(1)} \tag{46}$$

is (for the general case) given by

$$\begin{aligned} \Delta_{J\to b} &= \frac{\gamma \prod_i \left(\mu_{i\to J}(0) - \mu_{i\to J}(1)\right)}{\gamma \prod_i \left(\mu_{i\to J}(0) + \mu_{i\to J}(1)\right)} \qquad (47) \\ &= \prod_i \Delta_{i\to J}. \qquad (48) \end{aligned}$$

But by using

$$\Delta = \frac{\Lambda - 1}{\Lambda + 1} \tag{49}$$

and

$$\Lambda = \frac{1 + \Delta}{1 - \Delta}, \tag{50}$$

the desired check-to-bit update rule in terms of Λ consists in transforming all inputs $\Lambda_{i\to J}$ into normalized differences $\Delta_{i\to J}$, applying (48), and transforming $\Delta_{J\to b}$ back to $\Lambda_{J\to b}$.

Application to Log-Likelihood Ratios

The update rules of the sum-product algorithm take a particularly interesting form ([27, p. 83][16]) when written in terms of log-likelihoods

$$\begin{aligned}\lambda &\triangleq \log(\mu(0)/\mu(1)) & (51)\\ &= \log(\Lambda), & (52)\end{aligned}$$

where the logarithms are to the natural base e. The bit-to-check updating rule (45) then becomes that of the min-sum algorithm (40). From (49) and (52), the normalized difference can be expressed as

$$\begin{aligned}\Delta &= \frac{e^{\lambda}-1}{e^{\lambda}+1} & (53)\\ &= \frac{e^{\lambda/2}-e^{-\lambda/2}}{e^{\lambda/2}+e^{-\lambda/2}} & (54)\\ &= \tanh(\lambda/2), & (55)\end{aligned}$$

and the check-to-bit update rule (48) becomes

$$\lambda_{J,b} = 2\tanh^{-1}\left(\prod_i \tanh(\lambda_{i,J}/2)\right). \qquad (56)$$

Note that (41) may be viewed as an approximation of (56).

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] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

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

[30] 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.

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

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

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

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

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.