Preliminary versions of this paper were presented at ISIT 1997 in Ulm, Germany, on June 30, 1997, and at ISCTA 1997, Ambleside U.K., on July 15, 1997.

Draft of 23 Sep 1999 11:14 a.m.

The Generalized Distributive Law*

Srinivas M. Aji and Robert J. McEliece

Department of Electrical Engineering
California Institute of Technology
Pasadena, California 91125
(mas,rjm)@systems.caltech.edu

Abstract: In this semi-tutorial paper we discuss a general message-passing algorithm, which we call the generalized distributive law. The GDL is a synthesis of the work of many authors in the information theory, digital communications, signal processing, statistics, and artificial intelligence communities. It includes as special cases the Baum-Welch algorithm, the FFT on any finite Abelian group, the Gallager-Tanner-Wiberg decoding algorithm, Viterbi's algorithm, the BCJR algorithm, Pearl's "belief propagation" algorithm, the Shafer-Shenoy probability propagation algorithm, and the turbo decoding algorithm. Although this algorithm is guaranteed to give exact answers only in certain cases (the "junction tree" condition), unfortunately not including the cases of GTW with cycles or turbo-decoding, there is much experimental evidence, and a few theorems, suggesting that it often works approximately even when it is not supposed to.

Keywords: distributive law, graphical models, junction trees, belief propagation, turbo codes.

* This work was supported by NSF grant no. NCR-9505975, AFOSR grant no. 5F49620-97-1-0313, and grant from Qualcomm. A portion of McEliece's contribution was performed at the Sony Corporation in Tokyo, while he was a holder of a Sony Sabbatical Chair.

1. Introduction.

The humble distributive law, in its simplest form, states that $ab+ac = a(b+c)$. The left side of this equation involves three arithmetic operations (one addition and two multiplications), whereas the right side needs only two. Thus the distributive law gives us a "fast algorithm" for computing $ab + ac$. The object of this paper is to demonstrate that the distributive law can be vastly generalized, and that this generalization leads to a large family of fast algorithms, including Viterbi's algorithm and the FFT.

To give a better idea of the potential power of the distributive law and to introduce the viewpoint we shall take in this paper, we offer the following example.

1.1 Example. Let $f(x, y, w)$ and $g(x, z)$ be given real-valued functions, where x, y, z, and w are variables taking values in a finite set A with q elements. Suppose we are given the task of computing tables of the values of $\alpha(x, w)$ and $\beta(y)$, defined as follows:

$$\alpha(x, w) \stackrel{\text{def}}{=} \sum_{y,z \in A} f(x, y, w) g(x, z) \tag{1.1}$$

$$\beta(y) \stackrel{\text{def}}{=} \sum_{x,z,w \in A} f(x, y, w) g(x, z). \tag{1.2}$$

(We summarize (1.1) by saying that $\alpha(x, w)$ is obtained by "marginalizing out" the variables y and z from the function $f(x, y, w)g(x, z)$. Similarly, $\beta(y)$ is obtained by marginalizing out x, z, and w from the same function.) How many arithmetic operations (additions and multiplications) are required for this task? If we proceed in the obvious way, we notice that for each of the q^2 values of (x, w) there are q^2 terms in the sum defining $\alpha(x, w)$, each term requiring one addition and one multiplication, so that the total number of arithmetic operations required for the computation of $\alpha(x, w)$ is $2q^4$. Similarly, computing $\beta(y)$ requires $2q^4$ operations, so computing both $\alpha(x, w)$ and $\beta(y)$ using the direct method requires $4q^4$ operations.

On the other hand, because of the distributive law, the sum in (1.1) factors:

$$\alpha(x, w) = \left(\sum_{y \in A} f(x, y, w) \right) \cdot \left(\sum_{z \in A} g(x, z) \right). \tag{1.3}$$

Using this fact, we can simplify the computation of $\alpha(x, w)$. First we compute tables of the functions $\alpha_1(x, w)$ and $\alpha_2(x)$ defined by

$$\begin{aligned} \alpha_1(x, w) &\stackrel{\text{def}}{=} \sum_{y \in A} f(x, y, w) \\ \alpha_2(x) &\stackrel{\text{def}}{=} \sum_{z \in A} g(x, z), \end{aligned} \tag{1.4}$$

which requires a total of $q^3 + q^2$ additions. Then we compute the q^2 values of $\alpha(x, w)$ using the formula (cf. (1.3))

$$\alpha(x, w) = \alpha_1(x, w)\alpha_2(x), \tag{1.5}$$

which requires q^2 multiplications. Thus by exploiting the distributive law, we can reduce the total number of operations required to compute $\alpha(x, w)$ from $2q^4$ to q^3+2q^2. Similarly, the distributive law tells us that (1.2) can be written as

$$\begin{aligned}\beta(y) &= \sum_{x,w\in A} f(x,y,w)\left(\sum_{z\in A} g(x,z)\right) \\ &= \sum_{x,w\in A} f(x,y,w)\alpha_2(x),\end{aligned} \tag{1.6}$$

where $\alpha_2(x)$ is as defined in (1.4). Thus if we precompute a table of the values of $\alpha_2(x)$ (q^2 operations), and then use (1.6) ($2q^3$ further operations), we only need $2q^3 + q^2$ operations (as compared to $2q^4$ for the direct method) to compute the values of $\beta(y)$.

Finally, we observe that to compute *both* $\alpha(x, w)$ and $\beta(y)$ using the simplifications afforded by (1.3) and (1.6), we only need to compute $\alpha_2(x)$ once, which means that we can compute the values of $\alpha(x, w)$ and $\beta(y)$ with a total of only $3q^3 + 2q^2$ operations, as compared to $4q^4$ for the direct method.▪

The simplification in Example 1.1 was easy to accomplish, and the gains were relatively modest. In more complicated cases, it can be much harder to see the best way to reorganize the calculations, but the computational savings can be dramatic. It is the object of this paper to show that problems of the type described in Example 1.1 have a wide range of applicability, and to describe a general procedure, which we called the *generalized distributive law*, for solving them efficiently. Roughly speaking, the GDL accomplishes its goal by passing messages in a communications network whose underlying graph is a tree.

Important special cases of the GDL have appeared many times previously. In this paper, for example, we will demonstrate that the GDL includes as special cases the fast Hadamard transform, Viterbi's algorithm, the BCJR algorithm, the Gallager-Tanner-Wiberg decoding algorithm (when the underlying graph is cycle-free), and certain "probability propagation" algorithms known in the artificial intelligence community. With a little more work, we could have added the FFT on any finite Abelian group, the Baum-Welch "forward-backward" algorithm, and discrete-state Kalman filtering. Although this paper contains relatively little that is essentially new (for example, the 1990 paper of Shafer and Shenoy [33] describes an algorithm similar to the one we present in Section 3), we believe it is worthwhile to present a simply-stated algorithm of such wide applicability, which gives a unified treatment of a great many algorithms whose relationship to each other was not fully understood, if sensed at all.

Here is an outline of the paper. In Section 2, we will state a general computational problem we call the MPF ("marginalize a product function") problem, and show by example that a number of classical problems are instances of it. These problems include computing the discrete Hadamard transform, maximum-likelihood decoding of a linear code over a memoryless channel, probabilistic inference in Bayesian networks, a "probabilistic state machine" problem, and matrix chain multiplication. In Section 3, we shall give an exact algorithm for solving the MPF problem (the GDL) which often gives an efficient solution to the MPF problem. In Section 4 we will discuss the problem of finding junction trees (the formal name for the GDL's communication network), and "solve" the example instances of the MPF problem given in Section 2, thereby deriving, among other things, the fast Hadamard transform and Viterbi's algorithm. In Section 5 we will discuss the computational complexity of the GDL. (A proof of the correctness of the GDL is given in Appendix A.)

In Section 6, we give a brief history of the GDL. Finally, in Section 7, we speculate on the possible existence of an efficient class of approximate, iterative, algorithms for solving the MPF problem, obtained by allowing the communication network to have cycles. This speculation is based partly on the fact that two experimentally successful decoding algorithms, viz., the GTW algorithm for low-density parity-check codes, and the turbo decoding algorithm, can be viewed as an application of the GDL methodology on networks with cycles, and partly on some recent theoretical work on GDL-like algorithms on graphs with a single cycle.

Although this paper is semi-tutorial, it contains a number of things which have not appeared previously. Beside the generality of our exposition, these include:

• A de-emphasis of *a priori* graphical models, and an emphasis on algorithms to construct graphical models to fit the given problem.

• A number of non-probabilistic applications, including the FFT.

• A careful discussion of message scheduling, and a proof of the correctness of a large class of possible schedules.

• A precise measure of the computational complexity of the GDL.

2. The MPF Problem.

The GDL can greatly reduce the number of additions and multiplications required in a certain class of computational problems. It turns out that much of the power of the GDL is due to the fact that it applies to situations in which the notions of "addition" and "multiplication" are themselves generalized. The appropriate framework for this generalization is the commutative semiring.

Definition. A *commutative semiring* is a set K, together with two binary operations called "+" and "·", which satisfy the following three axioms:

S1. The operation "+" is associative and commutative, and there is an additive identity element called "0" such that $k + 0 = k$ for all $k \in K$. (This axiom makes $(K, +)$ a *commutative monoid*.)

S2. The operation "·" is also associative and commutative, and there is a multiplicative identity element called "1" such that $k \cdot 1 = k$ for all $k \in K$. (Thus $(K, \cdot)$ is also a commutative monoid.)

S3. The *distributive law* holds, i.e.,

$$(a \cdot b) + (a \cdot c) = a \cdot (b + c),$$

for all triples (a, b, c) from K. ▪

The difference between a semiring and a ring is that in a semiring, additive inverses need not exist, i.e., $(K, +)$ is only required to be a monoid, not a group. Thus every commutative ring is automatically a commutative semiring. For example, the set of real or complex numbers, with ordinary addition and multiplication, forms a commutative semiring. Similarly, the set of polynomials in one or more indeterminates over any commutative ring forms a commutative semiring. However, there are many other commutative semirings, some of which are summarized in Table 1. (In semirings 4–8, the set K is an interval of real numbers with the possible addition of $\pm\infty$.)

	K	"$(+, 0)$"	"$(\cdot, 1)$"	short name
1.	A	$(+, 0)$	$(\cdot, 1)$	
2.	$A[x]$	$(+, 0)$	$(\cdot, 1)$	
3.	$A[x, y, \ldots]$	$(+, 0)$	$(\cdot, 1)$	
4.	$[0, \infty)$	$(+, 0)$	$(\cdot, 1)$	sum-product
5.	$(0, \infty]$	$(\min, \infty)$	$(\cdot, 1)$	min-product
6.	$[0, \infty)$	$(\max, 0)$	$(\cdot, 1)$	max-product
7.	$(-\infty, \infty]$	$(\min, \infty)$	$(+, 0)$	min-sum
8.	$[-\infty, \infty)$	$(\max, -\infty)$	$(+, 0)$	max-sum
9.	$\{0, 1\}$	$(\texttt{OR}, 0)$	$(\texttt{AND}, 1)$	Boolean
10.	2^S	$(\cup, \emptyset)$	$(\cap, S)$	
11.	Λ	$(\vee, 0)$	$(\wedge, 1)$	
12.	Λ	$(\wedge, 1)$	$(\vee, 0)$.	

Table 1. Some commutative semirings. Here A denotes an arbitrary commutative ring, S is an arbitrary finite set, and Λ denotes an arbitrary distributive lattice.

For example, consider the min-sum semiring in Table 1 (entry 7). Here K is the set of real numbers, plus the special symbol "∞." The operation "+" is defined as the operation of *taking the minimum*, with the symbol ∞ playing the role of the corresponding identity element, i.e., we define $\min(k, \infty) = k$ for all $k \in K$. The operation "$\cdot$" is defined to be *ordinary addition* [sic], with the real number 0 playing the role of the identity, and $k + \infty = \infty$ for all k. Oddly enough, this combination forms a semiring, because the distributive law is equivalent to

$$\min(a + b, a + c) = a + \min(b, c),$$

which is easily seen to be true. We shall get a glimpse of the importance of this semiring in Examples 2.3 and 4.3, below. (In fact, semirings 5–8 are all isomorphic to each other;

for example 5 becomes 6 via the mapping $x \to 1/x$, and 6 becomes 7 under the mapping $x \to -\log x$.)

Having briefly discussed commutative semirings, we now describe the "marginalize a product function" problem, which is a general computational problem solved by the GDL. At the end of the section we will give several examples of the MPF problem, which demonstrate how it can occur in a surprisingly wide variety of settings.

Let $x_1, \ldots, x_n$ be variables taking values in the finite sets $A_1, \ldots, A_n$, with $|A_i| = q_i$ for $i = 1, \ldots, n$. If $S = \{i_1, \ldots, i_r\}$ is a subset of $\{1, \ldots, n\}$, we denote the product $A_{i_1} \times \cdots \times A_{i_r}$ by A_S, the variable list $(x_{i_1}, \ldots, x_{i_r})$ by x_S, and the cardinality of A_S, i.e., $|A_S|$, by q_S. We denote the product $A_{\{1,\ldots,n\}}$ simply by $\boldsymbol{A}$, and the variable list $\{x_1, \ldots, x_n\}$ simply by $\boldsymbol{x}$.

Now let $\mathcal{S} = \{S_1, \ldots, S_M\}$ be M subsets of $\{1, \ldots, n\}$. Suppose that for each $i = 1, \ldots, M$, there is a function $\alpha_i : A_{S_i} \to R$, where R is a commutative semiring. The variable lists x_{S_i} are called the *local domains* and the functions α_i are called the *local kernels*. We define the *global kernel* $\beta : \boldsymbol{A} \to R$, as follows:

$$\beta(x_1, \ldots, x_n) = \prod_{i=1}^{M} \alpha_i(x_{S_i}). \tag{2.1}$$

With this setup, the MPF problem is this: For one or more of the indices $i = 1, \ldots, M$, compute a table of the values of the S_i-*marginalization* of the global kernel β, which is the function $\beta_i : A_{S_i} \to R$, defined by

$$\beta_i(x_{S_i}) = \sum_{x_{S_i^c} \in A_{S_i^c}} \beta(\boldsymbol{x}). \tag{2.2}$$

In (2.2) S_i^c denotes the complement of the set S_i relative to the "universe" $\{1, \ldots, n\}$. For example, if $n = 4$, and if $S_i = \{1, 4\}$, then

$$\beta_i(x_1, x_4) = \sum_{x_2 \in A_2, x_3 \in A_3} \beta(x_1, x_2, x_3, x_4).$$

We will call the function $\beta_i(x_{S_i})$ defined in (2.2) the ith *objective function*, or the *objective function at* S_i. We note that the computation of the ith objective function in the obvious way requires $q_1 q_2 \cdots q_n$ additions and $(M-1) q_1 q_2 \cdots q_n$ multiplications, for a total of $M q_1 q_2 \cdots q_n$ arithmetic operations, where q_i denotes the size of the set A_i. We shall see below (Section 5) that the algorithm we call the "generalized distributive law" can often reduce this figure dramatically.

We conclude this section with some illustrative examples of the MPF problem.

2.1 Example. Let x_1, x_2, x_3, and x_4 be variables taking values in the finite sets A_1, A_2, A_3, and A_4. Suppose $f(x_1, x_2, x_4)$ and $g(x_1, x_3)$ are given functions of these variables, and that it is desired to compute tables of the functions $\alpha(x_1, x_4)$ and $\beta(x_2)$ defined by

$$\alpha(x_1, x_4) = \sum_{x_2 \in A_2, x_3 \in A_3} f(x_1, x_2, x_4) g(x_1, x_3)$$
$$\beta(x_2) = \sum_{x_1 \in A_1, x_3 \in A_3, x_4 \in A_4} f(x_1, x_2, x_4) g(x_1, x_3).$$

This is an instance of the MPF problem, if we define local domains and kernels as follows:

	local domain	local kernel
1.	$\{x_1, x_2, x_4\}$	$f(x_1, x_2, x_4)$
2.	$\{x_1, x_3\}$	$g(x_1, x_3)$
3.	$\{x_1, x_4\}$	1
4.	$\{x_2\}$	1

The desired function $\alpha(x_1, x_4)$ is the objective function at local domain 3, and $\beta(x_2)$ is the objective function at local domain 4. This is just a slightly altered version of Example 1.1, and we shall see in Section 4 that when the GDL is applied, the "algorithm" of Example 1.1 results. ▪

2.2 Example. Let x_1, x_2, x_3, y_1, y_2, and y_3 be six variables, each assuming values in the binary set $\{0, 1\}$, and let $f(y_1, y_2, y_3)$ be a real-valued function of the variables y_1, y_2, and y_3. Now consider the MPF problem (the commutative semiring being the set of real numbers with ordinary addition and multiplication) with the following local domains and kernels.

	local domain	local kernel
1.	$\{y_1, y_2, y_3\}$	$f(y_1, y_2, y_3)$
2.	$\{x_1, y_1\}$	$(-1)^{x_1 y_1}$
3.	$\{x_2, y_2\}$	$(-1)^{x_2 y_2}$
4.	$\{x_3, y_3\}$	$(-1)^{x_3 y_3}$
5.	$\{x_1, x_2, x_3\}$	1.

Here the global kernel, i.e., the product of the local kernels, is

$$F(x_1, x_2, x_3, y_1, y_2, y_3) = f(y_1, y_2, y_3)(-1)^{x_1 y_1 + x_2 y_2 + x_3 y_3},$$

and the objective function at the local domain $\{x_1, x_2, x_3\}$ is

$$F(x_1, x_2, x_3) = \sum_{y_1, y_2, y_3} f(y_1, y_2, y_3)(-1)^{x_1 y_1 + x_2 y_2 + x_3 y_3},$$

which is the *Hadamard transform* of the original function $f(y_1, y_2, y_3)$ [17]. Thus the problem of computing the Hadamard transform is a special case of the MPF problem. (A straightforward generalization of this example shows that the problem of computing the Fourier transform over any finite Abelian group is also a special case of the MPF problem. While the kernel for the Hadamard transform is of diagonal form, in general, the kernel will only be lower triangular. See [1, Chapter 3] for the details.) We shall see below in Example 4.2 that the GDL algorithm, when applied to this set of local domains and kernels, yields the *fast* Hadamard transform. ▪

2.3 Example (Wiberg [39]). Consider the $(7,4,2)$ binary linear code defined by the parity-check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Suppose that an unknown codeword $(x_1, x_2, \ldots, x_7)$ from this code is transmitted over a discrete memoryless channel, and that the vector $(y_1, y_2, \ldots, y_7)$ is received. The "likelihood" of a particular codeword $(x_1, x_2, \ldots, x_7)$ is then

$$p(y_1, \ldots, y_7 | x_1, \ldots, x_7) = \prod_{i=1}^{7} p(y_i | x_i), \tag{2.3}$$

where the $p(y_i|x_i)$'s are the transition probabilities of the channel. The *maximum likelihood decoding problem* is that of finding the codeword that maximizes the expression in (2.3). Now consider the MPF problem with the following domains and kernels, using the min-sum semiring (semiring 7 from Table 1). There is one local domain for each codeword coordinate and one for each row of the parity-check matrix.

	local domain	local kernel
1.	$\{x_1\}$	$-\log p(y_1\|x_1)$
$\vdots$	$\vdots$	
7.	$\{x_7\}$	$-\log p(y_7\|x_7)$
8.	$\{x_1, x_2, x_4\}$	$\chi(x_1, x_2, x_4)$
9.	$\{x_3, x_4, x_6\}$	$\chi(x_3, x_4, x_6)$
10.	$\{x_4, x_5, x_7\}$	$\chi(x_4, x_5, x_7)$

Here χ is a function that indicates whether a given parity-check is satisfied, or not. For example, at the local domain $\{x_1, x_2, x_4\}$, which corresponds to the first row of the parity-check matrix, we have

$$\chi(x_1, x_2, x_4) = \begin{cases} 0 & \text{if } x_1 + x_2 + x_4 = 0 \\ \infty & \text{if } x_1 + x_2 + x_4 = 1. \end{cases}$$

The global kernel is then

$$F(x_1, \ldots, x_7) = \begin{cases} -\log p(y_1, \ldots, y_7 | x_1, \ldots, x_7) & \text{if } (x_1, \ldots, x_7) \text{ is a codeword} \\ \infty & \text{if } (x_1, \ldots, x_7) \text{ is not a codeword.} \end{cases}$$

Thus the objective function at the local domain $\{x_i\}$ is

$$F_i(a_i) = \min\{-\log p(y_1, \ldots, y_7 | x_1, \ldots, x_7) : \text{all codewords for which } x_i = a.\}$$

It follows that the value of a_i for which $F_i(a_i)$ is smallest is the value of the ith component of a maximum likelihood codeword, i.e., a codeword for which $p(y_1, \ldots, y_7|x_1, \ldots, x_7)$ is largest. A straightforward extension of this example shows that the problem of maximum-likelihood decoding of an arbitrary linear block code is a special case of the MPF problem. We shall see in Example 4.3 that when the GDL is applied to problems of this type, the Gallager-Tanner-Wiberg decoding algorithm results. ▪

2.4 Example. Consider the directed acylic graph (DAG) G in Figure 1.[1] In a DAG, the "parents" of a vertex v, denoted $\mathrm{pa}(v)$, are those vertices (if any) which lie immediately "above" v. Thus in Figure 1, $\mathrm{pa}(A) = \{B, E\}$, and $\mathrm{pa}(B) = \emptyset$. Let us associate a random variable with each of the vertices, and assume that each random variable is dependent only on its "parents," i.e., the joint density function factors as follows:

$$\Pr\{B = b, E = e, A = a, R = r, W = w\} =$$
$$\Pr\{B = b\} \Pr\{E = e\} \Pr\{A = a|B = b, E = e\} \Pr\{R = r|E = e\} \Pr\{W = w|A = a\},$$

or, using streamlined notation,

$$p(b, e, a, r, w) = p(b)p(e)p(a|b, e)p(r|e)p(w|a). \tag{2.4}$$

A DAG, together with associated random variables whose joint density function factors according to the structure of the DAG, is called a *Bayesian network* [18].

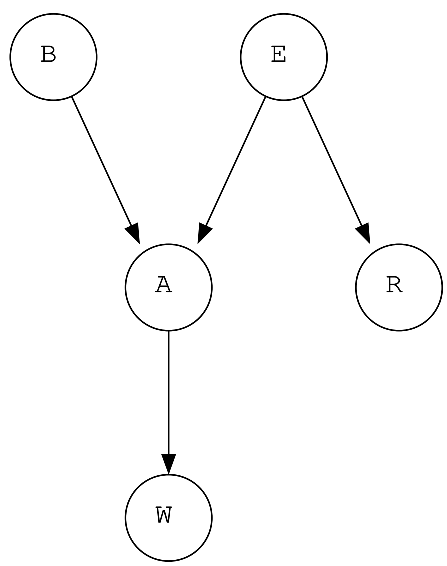

Figure 1 . The Bayesian network in Example 2.4.

[1] This example is taken from [29], in which B stands for burglary, E is for earthquake, A is for alarm sound, R is for radio report, and W is for Watson's call.

Let us assume that the two random variables W and R are observed to have the values w_0 and r_0, respectively. The *probabilistic inference problem*, in this case, is to compute the conditional probabilities of one or more of the remaining random variables, i.e., B, E, and A, where the conditioning is with respect to the "evidence" $\{R = r_0, W = w_0\}$. Now consider the MPF problem with the following local domains and kernels.

	local domain	local kernel
1.	$\{b\}$	$p(b)$
2.	$\{e\}$	$p(e)$
3.	$\{a, b, e\}$	$p(a\|b, e)$
4.	$\{a\}$	$p(w_0\|a)$
5.	$\{e\}$	$p(r_0\|e)$

Then by (2.4) (using semiring 4 from Table 1, viz. the set of nonnegative real numbers with ordinary addition and multiplication) the global kernel $F(b, e, a)$ is just the function $p(b, e, a, r_0, w_0)$, so that, for example, the objective function at local domain 1 is

$$\begin{aligned} F_1(b) &= \sum_{e,a} p(b, e, a, r_0, w_0) \\ &= p(b, r_0, w_0). \end{aligned}$$

But by Bayes' rule, $p(b|r_0, w_0) = p(b, r_0, w_0)/p(w_0, r_0)$, so that the conditional probability of B, given the "evidence" (r_0, w_0), is

$$\Pr\{B = b | R = r_0, W = w_0\} = p(b|r_0, w_0) = \alpha F_1(b),$$

where the constant of proportionality α is given by

$$\alpha = \left(\sum_b F_1(b) \right)^{-1}.$$

Similarly, the computation of the conditional probabilities of A and E can be accomplished via evaluation of the objective functions at the local domains 4 and 5, respectively. Thus the problem of probabilistic inference in Bayesian networks is a special case of the MPF problem. We shall see in Section 4 that when the GDL is applied to problems of this type, the result is an algorithm equivalent to the "probability propagation" algorithms known in the artificial intelligence community. ▪

2.5 Example. As a more useful instance of the probabilistic inference problem, we consider a *probabilistic state machine.*[2] At each time $t \in \{0, 1, \ldots, n-1\}$, the PSM has state s_t, input u_t and output y_t. The u_t are probabilistically generated, independently, with probabilities $p(u_t)$. The output y_t depends on the state s_t and input u_t and is described

[2] Our probabilistic state machines are closely related to the "hidden Markov models" considered in the literature [32].

by the conditional probability distribution $p(y_t|s_t, u_t)$. The state s_{t+1} also depends on s_t and u_t, with conditional probability $p(s_{t+1}|s_t, u_t)$. If the *a priori* distribution of the initial state s_0 is known, we can write the joint probability of the inputs, state and outputs from time 0 to $n-1$ as

$$\begin{aligned}(2.5)\qquad & p(u_0, \ldots, u_{n-1}, s_0, \ldots, s_{n-1}, y_0, \ldots, y_{n-1}) = \\ & p(s_0)p(u_0)p(y_0|s_0, u_0) \prod_{t=1}^{n-1} p(s_t|s_{t-1}, u_{t-1})p(u_t)p(y_t|s_t, u_t).\end{aligned}$$

This means that the PSM is a Bayesian network, as depicted in Figure 2.

Suppose we observe the n output values, denoting these observations by $y_0^e, \ldots, y_{n-1}^e$ ("e" for evidence), and wish to infer the values of the inputs based on this evidence. This is a probabilitic inference problem of the type discussed in Example 2.4. We can compute the conditional probability $\Pr\{u_t = a|y_0^e, \ldots, y_{n-1}^e\}$ by taking the joint probability in (2.5) with the observed values of y_t^e and marginalizing out all the s_t's and all but one of the u_t's. This is an instance of the MPF problem, with the following local domains and kernels (illustrated for $n = 4$):

	local domain	local kernel
1.	$\{u_0\}$	$p(u_0)$
2.	$\{u_1\}$	$p(u_1)$
3.	$\{u_2\}$	$p(u_2)$
4.	$\{u_3\}$	$p(u_3)$
5.	$\{s_0\}$	$p(s_0)$
6.	$\{u_0, s_0\}$	$p(y_0^e|u_0, s_0)$
7.	$\{u_1, s_1\}$	$p(y_1^e|u_1, s_1)$
8.	$\{u_2, s_2\}$	$p(y_2^e|u_2, s_2)$
9.	$\{u_3, s_3\}$	$p(y_3^e|u_3, s_3)$
10.	$\{u_0, s_0, s_1\}$	$p(s_1|u_0, s_0)$
11.	$\{u_1, s_1, s_2\}$	$p(s_2|u_1, s_1)$
12.	$\{u_2, s_2, s_3\}$	$p(s_3|u_2, s_2)$

This model includes as a special case convolutional codes, as follows. The state transition is deterministic, which means that $p(s_{t+1}|s_t, u_t) = 1$ when $s_{t+1} = s_{t+1}(s_t, u_t)$ and $p(s_{t+1}|s_t, u_t) = 0$ otherwise. Assuming a memoryless channel, the output y_t is probabilistically dependent on x_t, which is a deterministic function of the state and input, and so $p(y_t|s_t, u_t) = p(y_t|x_t(s_t, u_t))$. Marginalizing the product of functions in (2.5) in the sum-product and max-product semirings will then give us the maximum-likelihood input *symbols* or input *block*, respectively. As we shall see below (Example 4.5), when the GDL is applied here, we get algorithms equivalent to the BCJR and Viterbi decoding algorithms.[3]
■

[3] Technically, to obtain an algorithm equivalent to Viterbi's, it is necessary to take the negative logarithm of (2.5), and then perform the marginalization in the min-sum semiring.

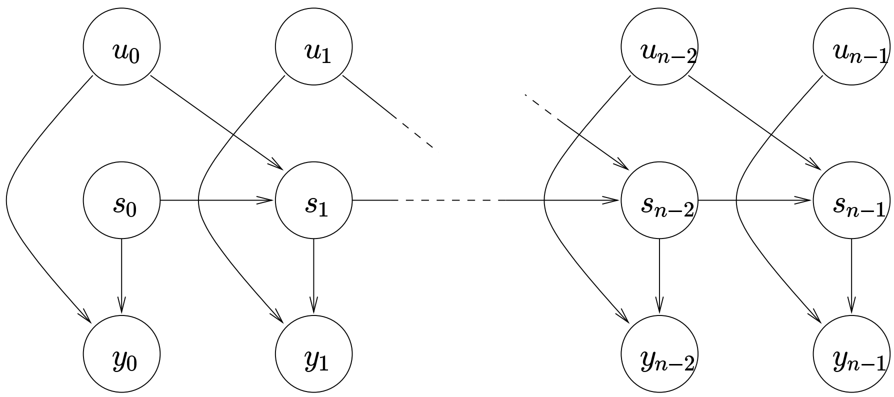

Figure 2 . The Bayesian network for the probabilistic state machine in Example 2.5.

2.6 Example. Let M_i be a $q_{i-1} \times q_i$ matrix with entries in a commutative semiring, for $i = 1, \ldots, n$. We denote the entries in M_i by $M_i[x_{i-1}, x_i]$, where for $i = 0, 1, \ldots, n$, x_i is a variable taking values in a set A_i with q_i elements. Suppose we want to compute the product

$$M = M_1 \cdot M_2 \cdots M_n.$$

Then for $n = 2$ we have by definition

$$M[x_0, x_2] = \sum_{x_1} M_1[x_0, x_1] M_2[x_1, x_2], \tag{2.6}$$

and an easy induction argument gives the generalization

$$M[x_0, x_n] = \sum_{x_1, \ldots, x_{n-1}} M_1[x_0, x_1] \cdots M_n[x_{n-1}, x_n]. \tag{2.7}$$

(Note that (2.7) suggests that the number of arithmetic operations required to multiply these n matrices is $2q_0 q_1 \cdots q_n$.) Thus multiplying n matrices can be formulated as the following MPF problem:.

	local domain	local kernel
1.	$\{x_0, x_1\}$	$M_1[x_0, x_1]$
2.	$\{x_1, x_2\}$	$M_2[x_1, x_2]$
	$\vdots$	
n	$\{x_{n-1}, x_n\}$	$M_n[x_{n-1}, x_n]$
$n+1$	$\{x_0, x_n\}$	1,

the desired result being the objective function at local domain $n+1$.

As an alternative interpretation of (2.7), consider a trellis of depth n, with vertex set $A_0 \cup A_1 \cdots \cup A_n$, and an edge of weight $M_i[x_{i-1}, x_i]$ connecting the vertices $x_{i-1} \in A_{i-1}$ and $x_i \in A_i$. If we define the weight of a path as the sum of the weights of the component edges, then $M[x_0, x_n]$ as defined in (2.7) represents the sum of the weights of all paths from x_0 to x_n. For example, Figure 3 shows the trellis corresponding to the multiplication of three matrices, of sizes 2×3, 3×3, and 3×2. If the computation is done in the min-sum semiring, the interpretation is that $M[x_0, x_n]$ is the weight of a minimum-weight path from x_0 to x_n.

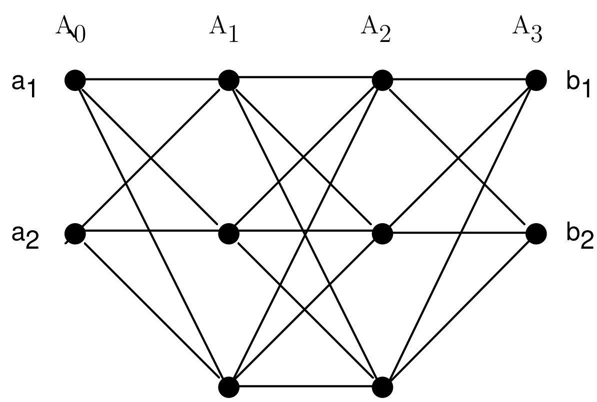

Figure 3 . The trellis corresponding to the multiplication of three matrices, of sizes 2×3, 3×3, and 3×2. The (i, j)th entry in the matrix product is the sum of the weights of all paths from a_i to b_j.

We shall see in Section 4 that if the GDL is applied to the matrix multiplication problem, a number of different algorithms result, corresponding to the different ways of parenthesizing the expression $M_1 M_2 \cdots M_n$. If the parenthesization is

$$(((M_1 M_2)M_3)M_4)M_5,$$

(illustrated for $n = 5$), and the computation is in the min-sum semiring, Viterbi's algorithm results. ▪

3. The GDL: an algorithm for solving the MPF problem.

If the elements of $\mathcal{S}$ stand in a certain special relationship to each other, then an algorithm for solving the MPF problem can be based on the notion of "message passing." The required relationship is that the local domains can be organized into a *junction tree* [18]. What this means is that the elements of $\mathcal{S}$ can be attached as labels to the vertices of a graph-theoretic tree T, such that for any two vertices v_i and v_j, the intersection of the corresponding labels, viz., $S_i \cap S_j$, is a subset of the label on each vertex on the unique path from v_i to v_j. Alternatively, the subgraph of T consisting of those vertices whose label includes the element i, together with the edges connecting these vertices, is connected, for $i = 1, \ldots, n$.

For example, consider the following five local domains:

	local domain
1.	$\{x_1\}$
2.	$\{x_1, x_2\}$
3.	$\{x_1, x_3\}$
4.	$\{x_2\}$
5.	$\{x_2, x_4\}$

These local domains can be organized into a junction tree, as shown in Figure 4. For example, the unique path from vertex 2 to vertex 3 is $v_2 \to v_1 \to v_3$, and $S_2 \cap S_3 \subseteq S_1$, as required.

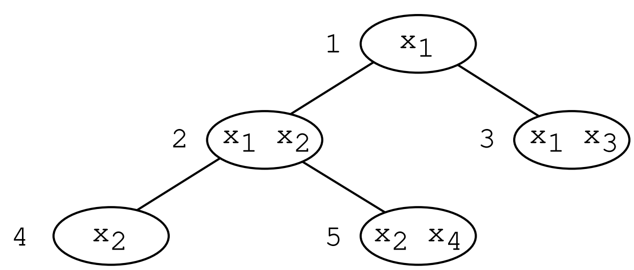

Figure 4 . A junction tree.

On the other hand, the following set of four local domains cannot be organized into a junction tree, as can be easily verified.

	local domain
1.	$\{x_1, x_2\}$
2.	$\{x_2, x_3\}$
3.	$\{x_3, x_4\}$
4.	$\{x_1, x_4\}$

However, by adjoining two “dummy domains”

	local domain
5.	$\{x_1, x_2, x_4\}$
6.	$\{x_2, x_3, x_4\}$

to the collection, we can devise a junction tree, as shown in Figure 5.

(In Section 4, below, we give a simple algorithm for deciding whether or not a given set of local domains can be organized into a junction tree, for constructing one if it does exist, and for finding appropriate dummy domains if it does not.)

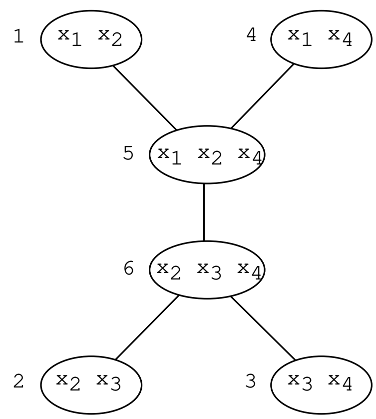

Figure 5 . A junction tree which includes the local domains $\{x_1, x_2\}$, $\{x_2, x_3\}$, $\{x_3, x_4\}$, and $\{x_1, x_4\}$.

In the “junction tree” algorithm, which is what we call the *generalized distributive law* (GDL), if v_i and v_j are connected by an edge (indicated by the notation $v_i \operatorname{adj} v_j$), the (directed) “message” from v_i to v_j is a table containing the values of a function $\mu_{i,j} : A_{S_i \cap S_j} \to R$. Initially, all such functions are defined to be identically 1 (the semiring’s multiplicative identity); and when a particular message $\mu_{i,j}$ is updated, the following rule is used:

$$\mu_{i,j}(x_{S_i \cap S_j}) = \sum_{x_{S_i \setminus S_j} \in A_{S_i \setminus S_j}} \alpha_i(x_{S_i}) \prod_{\substack{v_k \operatorname{adj} v_i \\ k \neq j}} \mu_{k,i}(x_{S_k \cap S_i}). \tag{3.1}$$

A good way to remember (3.1) is to think of the junction tree as a communication network, in which an edge from v_i to v_j is a transmission line that “filters out” dependence on all variables but those common to v_i and v_j. (The filtering is done by marginalization.) When the vertex v_i wishes to send a message to v_j, it forms the product of its local kernel with all messages it has received from its neighbors other than v_j, and transmits the product to v_j over the (v_i, v_j) transmission line.

Similarly the "state" of a vertex v_i is defined to be a table containing the values of a function $\sigma_i : A_{S_i} \to R$. Initially, σ_i is defined to be the local kernel $\alpha_i(x_{S_i})$, but when σ_i is updated, the following rule is used.

$$\sigma_i(x_{S_i}) = \alpha_i(x_{S_i}) \prod_{v_k \text{ adj } v_i} \mu_{k,i}(x_{S_k \cap S_i}). \tag{3.2}$$

In words, the state of a vertex v_i is the product of its local kernel with each of the messages it has received from its neighbors. The basic idea is that after sufficiently many messages have been passed, $\sigma_i(x_{S_i})$ will be the objective function at S_i, as defined in (2.2).

The question remains as to the scheduling of the message-passing and the state computation. Here we consider two special cases, the *single-vertex* problem, in which the goal is to compute the objective function at only one vertex v_0, and the *all-vertices* problem, where the goal is to compute the objective function at all vertices.[4]

For the single-vertex problem, the natural (serial) scheduling of the GDL begins by directing each edge toward the target vertex v_0. Then messages are sent only in the direction toward v_0, and each directed message is sent only once. A vertex sends a message to a neighbor, when, for the first time, it has received messages from each of its other neighbors. The target v_0 computes its state when it has received messages from each of its neighbors. With this scheduling, messages begin at the leaves (vertices with degree 1), and proceed toward v_0, until v_0 has received messages from all its neighbors, at which point v_0 computes its state and the algorithm terminates.

For example, if we wish to solve the single-vertex problem for the junction tree of Figure 4, and the target vertex is v_1, the edges should all be directed towards v_1, as shown in Figure 6. Then one possible sequence of messages and state computations runs as shown in Table 2.

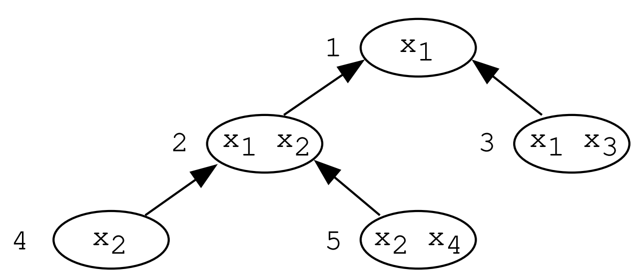

Figure 6 . The junction tree of Figure 4,
with the edges directed towards v_1.

[4] We do not consider the problem of evaluating the objective function at k vertices, where $1 < k < M$. However, as we will see in Section 5, the complexity of the M-vertex GDL is at most four times as large as the 1-vertex GDL, so it is reasonably efficient to "solve" the k-vertex problem using the M-vertex solution.

Round	Message or State Computation
1	$\mu_{3,1}(x_1) = \sum_{x_3} \alpha_3(x_1, x_3)$
2	$\mu_{4,2}(x_2) = \alpha_4(x_2)$
3	$\mu_{5,2}(x_2) = \sum_{x_4} \alpha_5(x_2, x_4)$
4	$\mu_{2,1}(x_1) = \sum_{x_2} \alpha_2(x_1, x_2) \cdot \mu_{4,2}(x_2) \cdot \mu_{5,2}(x_2)$
5	$\sigma_1(x_1) = \alpha_1(x_1) \cdot \mu_{2,1}(x_1) \cdot \mu_{3,1}(x_1).$

Table 2.
A schedule for the single-vertex GDL for the junction tree of Figure 4, with target vertex v_1.

It will be shown in Section 5 that this scheduling of the single vertex GDL requires at most

$$\sum_v d(v)|A_{S(v)}| \qquad \text{arithmetic operations,} \tag{3.3}$$

where $S(v)$ is the label of v, and $d(v)$, the *degree* of v, is the number of vertices adjacent to v. This should be compared to the complexity of the "obvious" solution, which as we noted above is $Mq_1 \cdots q_n$ operations. For example, for the junction tree shown in Figure 6, the complexity of the single-vertex GDL is by (3.3) at most $2q_1 + 3q_1q_2 + q_1q_3 + q_2 + q_2q_4$ arithmetic operations, vs. $5q_1q_2q_3q_4q_4$ for the direct computation.

For the all-vertices problem, the GDL can be scheduled in several ways. For example, in a fully parallel implementation, at every iteration, every state is updated, and every message is computed and transmitted, simultaneously. In this case the messages and states will stabilize after a number of iterations at most equal to the diameter of the tree, at which point the states of the vertices will be equal to the desired objective functions, and the algorithm terminates. Alternatively, the GDL can be scheduled fully serially, in which case each message is sent only once, and each state is computed only once. In this case, a vertex sends a message to a neighbor when, for the first time, it has received messages from all of its other neighbors, and computes its state when, for the first time, it has received messages from all its neighbors. In this serial mode, messages begin at the leaves, and proceed inwards into the tree, until some nodes have received messages from all their neighbors, at which point messages propagate outwards, so that each vertex eventually receives messages from all of its neighbors.[5] We will see in Section 4 that the fully-serial all-vertices GDL requires at most $4\sum_{v\in V} d(v)|A_{S(v)}|$ arithmetic operations.

There are also a variety of possible hybrid schedules, intermediate between fully parallel and fully serial. For example, Table 3 shows a hybrid schedule for the junction tree

[5] We might therefore call the fully serial all-vertices GDL an "inward-outward" algorithm.

of Figure 4, in which the computation is organized into four rounds. The computations in each round may be performed in any order, or even simultaneously, but the rounds must be performed sequentially.

Round	Message and State Computations
1	$\mu_{3,1}(x_1) = \sum_{x_3} \alpha_3(x_1, x_3)$
	$\mu_{4,2}(x_2) = \alpha_4(x_2)$
	$\mu_{5,2}(x_2) = \sum_{x_4} \alpha_5(x_2, x_4)$
2	$\mu_{1,2}(x_1) = \sum_{x_2} \alpha_1(x_1, x_2) \cdot \mu_{3,1}(x_1)$
	$\mu_{2,1}(x_1) = \sum_{x_2} \alpha_2(x_1, x_2) \cdot \mu_{4,2}(x_2) \cdot \mu_{5,2}(x_2)$
3	$\mu_{1,3}(x_1) = \alpha_1(x_1) \cdot \mu_{2,1}(x_1)$
	$\mu_{2,4}(x_2) = \sum_{x_1} \alpha_2(x_1, x_2) \cdot \mu_{1,2}(x_1) \cdot \mu_{5,2}(x_2)$
	$\mu_{2,5}(x_2) = \sum_{x_1} \alpha_2(x_1, x_2) \cdot \mu_{1,2}(x_1) \cdot \mu_{4,2}(x_2)$
	$\sigma_1(x_1) = \alpha_1(x_1) \cdot \mu_{2,1}(x_1) \cdot \mu_{3,1}(x_1)$
	$\sigma_2(x_1, x_2) = \alpha_2(x_1, x_2) \cdot \mu_{1,2}(x_1, x_2) \cdot \mu_{4,2}(x_2) \cdot \mu_{5,2}(x_2)$
4	$\sigma_3(x_1, x_3) = \alpha_3(x_1, x_3) \cdot \mu_{1,3}(x_1)$
	$\sigma_4(x_2) = \alpha_4(x_2) \cdot \mu_{2,4}(x_2)$
	$\sigma_5(x_2, x_4) = \alpha_5(x_2, x_4) \cdot \mu_{2,5}(x_2)$.

Table 3.
A "hybrid" schedule for the all-vertices GDL for the junction tree of Figure 4.

That concludes our informal discussion of GDL scheduling. We end this section with what we call the "Scheduling Theorem" for the GDL.

Thus let T be a junction tree with vertex set V and edge set E. In the GDL, messages can be passed in both directions on each edge, so it will be convenient to regard the edge set E as consisting of ordered pairs of vertices. Thus for example for the tree of Figure 4, we have

$$E = \{(1,2), (2,1), (1,3), (3,1), (2,4), (4,2), (2,5), (5,2)\}.$$

A *schedule* for the GDL is defined to be a finite sequence of subsets of E. A typical schedule will be denoted by $\mathcal{E} = (E_1, E_2, \ldots, E_N)$. The idea is that E_t is the set of messages that are updated during the tth round of the algorithm. In Tables 2 and 3, for example, the corresponding schedules are

Table 2: $\quad \mathcal{E} = (\{(3,1)\}, \{(4,2)\}, \{(5,2)\}, \{(2,1)\})$

Table 3: $\quad \mathcal{E} = (\{(3,1), (4,2), (5,2)\}, \{(1,2), (2,1)\}, \{(1,3), (2,4), (2,5)\})$.

Given a schedule $\mathcal{E} = (E_1, E_2, \ldots, E_N)$, the corresponding *message trellis* is a finite directed graph with vertex set $V \times \{0, 1, \ldots, N\}$, in which a typical element is denoted by $v_i(t)$, for $t \in \{0, 1, \ldots, N\}$. The only allowed edges are of the form $(v_i(t-1), v_j(t))$; and $(v_i(t-1), v_j(t))$ is an edge in the message trellis if either $(v_i, v_j) \in E_t$ or $i = j$. The message trellises for the junction tree of Figure 4, under the schedules of Tables 2 and 3, are shown in Figures 7 and 8, respectively. (In these figures, the shaded boxes indicate which local kernels are known to which vertices at any time. For example, in Figure 7, we can see that knowledge of the local kernels α_2, α_4, and α_5 has reached v_2 at time $t = 3$. We will elaborate on this notion of "knowldege" in Appendix A.)

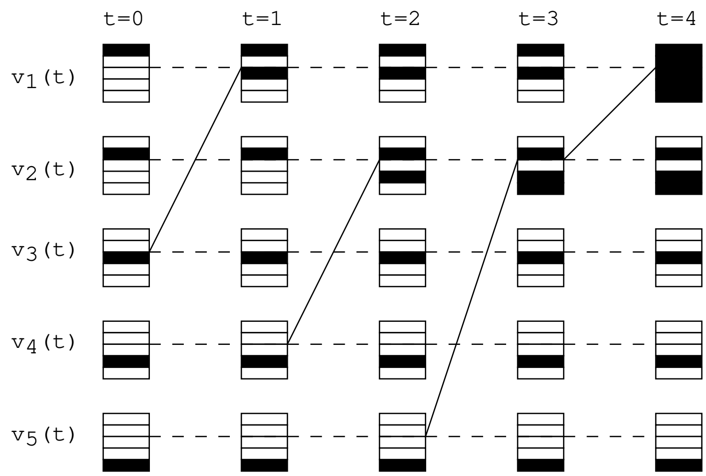

Figure 7 . The message trellis for the junction tree in Figure 4, under the schedule of Table 2, viz., $E_1 = \{(3,1)\}$, $E_2 = \{(4,2)\}$, $E_3 = \{(5,2)\}$, $E_4 = \{(2,1)\}$.

3.1 Theorem (GDL Scheduling). *After the completion of the message passing described by the schedule $\mathcal{E} = (E_1, E_2, \ldots, E_N)$, the state at vertex v_j will be the jth objective as defined in (3.2) if and only if there is a path from $v_i(0)$ to $v_j(N)$ in the corresponding message trellis, for $i = 1, \ldots, n$.*

A proof of Theorem 3.1 will be found in Appendix A, but Figures 7 and 8 illustrate the idea. For example, in Figure 8, we see that there is a path from each of $v_1(0), \ldots, v_5(0)$ to $v_2(2)$, which means (by the scheduling theorem) that after two rounds of message passing, the state at v_2 will be the desired objective function. This is why, in Table 3, we are able to compute σ_2 in round 3. Theorem 3.1 immediately implies the correctness of the single vertex and all-vertices serial GDL described earlier in this section.

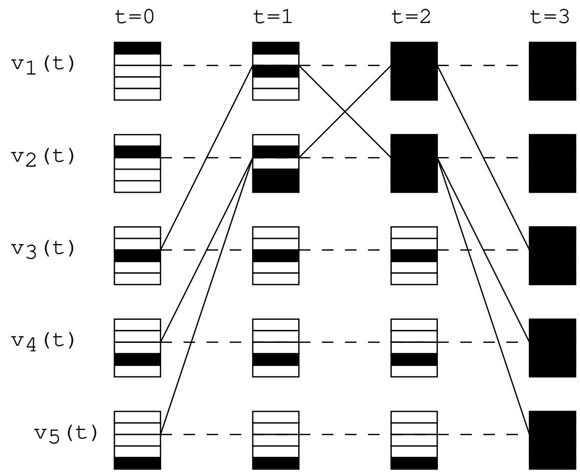

Figure 8 . The message trellis for the junction tree in Figure 4, under the schedule of Table 3, viz., $E_1 = \{(3,1),(4,2),(5,2)\}$, $E_2 = \{(1,2),(2,1)\}$, $E_3 = \{(1,3),(2,4),(2,5)\}$.

4. Constructing Junction Trees.

In Section 3 we showed that if we can construct a junction tree with the local domains as vertex labels, we can devise a message-passing algorithm to solve the MPF problem. But does such a junction tree exist? And if not, what can be done? In this section we will answer these questions.

It is easy to decide whether or not a junction tree exists. The key is the *local domain graph* G_{LD}, which is a weighted complete graph with M vertices $v_1, \ldots, v_M$, one for each local domain, with the weight of the edge $e_{i,j} : v_i \leftrightarrow v_j$ defined by

$$w_{i,j} = |S_i \cap S_j|.$$

If $x_k \in S_i \cap S_j$, we will say that x_k is contained in $e_{i,j}$. Denote by $w_{\max}$ the weight of a maximal-weight spanning tree of G_{LD}.[6] Finally, define

$$w^* = \sum_{i=1}^{M} |S_i| - n.$$

4.1 Theorem. $w_{\max} \le w^*$, *with equality if and only if there is a junction tree. If* $w_{\max} = w^*$, *then any maximal-weight spanning tree of* G_{LD} *is a junction tree.*

Proof: For each $k = 1, \ldots, n$, denote by m_k the number of sets S_i which contain the variable x_k. Note that $\sum_{k=1}^{n} m_k = \sum_{i=1}^{M} |S_i|$. Let T be any spanning tree of G_{LD}, and

[6] A maximal-weight spanning tree can easily be found with Prim's "greedy" algorithm [27], Chapter 3, [9], Section 24.2. In brief, Prim's algorithm works by growing the tree one edge at a time, always choosing a new edge of maximal weight.

let w_k denote the number of edges in T which contain x_k. Clearly $w(T) = \sum_{k=1}^n w_k$. Furthermore, $w_k \le m_k - 1$, since the subgraph T_k of T induced by the vertices containing x_k has no cycles, and equality holds if and only if T_k is connected, i.e., a tree. It follows then that

$$w(T) = \sum_{k=1}^{n} w_k \le \sum_{k=1}^{n} (m_k - 1) = \sum_{i=1}^{M} |S_i| - n = w^*,$$

with equality if and only if each subgraph T_k is connected, i.e., if T is a junction tree. ▪

4.1 Example. Here we continue Example 2.1. The LD graph is shown in Figure 9 (a). Here $w^* = (3+2+2+1)-4 = 4$. A maximal weight spanning tree is shown in Figure 9 (b), and its weight is 4, so by Theorem 4.1, this is a junction tree, a fact that can easily be checked directly. If we apply the GDL to this junction tree, we get the "algorithm" described in our introductory example Example 1.1 (if we use the schedule $\mathcal{E} = \{E_1, E_2, E_3\}$, where $E_1 = \{(4,1),(2,3)\}$, $E_2 = \{(1,3),(3,1)\}$, $E_3 = \{(1,4)\}$). ▪

If no junction tree exists with the given vertex labels S_i, all is not lost. We can always find a junction tree with M vertices such that each S_i is a *subset* of the ith vertex label, so that each local kernel α_i may be associated with the ith vertex, by regarding it as a function of the variables involved in the label. The key to this construction is the *moral graph*[7] which is the undirected graph with vertex set equal to the set of variables $\{x_1, \ldots, x_n\}$, and having an edge between x_i and x_j if there is a local domain which contains both x_i and x_j.

Given a cycle in a graph, a *chord* is an edge between two vertices on the cycle which do not appear consecutively in the cycle. A graph is *triangulated* if every simple cycle (i.e., one with no repeated vertices) of length larger than three has a chord.

In [18], it is shown that the cliques (maximal complete subgraphs) of a graph can be the vertex labels of a junction tree if and only if the graph is triangulated. Thus, to form a junction tree with vertex labels such that each of the local domains is contained in some vertex label, we form the moral graph, add enough edges to the moral graph so that the resulting graph is triangulated, and then form a junction tree with the cliques of this graph as vertex labels. Each of the original local domains will be a subset of at least one of these cliques. We can then attach the original local domains as "leaves" to the clique junction tree, thereby obtaining a junction tree for the original set of local domains and kernels, plus "extra" local domains corresponding to the cliques in the moral graph. We can then associate one of the local kernels attached to each of the cliques to that clique, and delete the corresponding leaf. In this way we will have constructed a junction tree for the original set of local kernels, with some of the local domains enlarged to include extra variables. However, this construction is far from unique, and the choices that must be made (which edges to add to the moral graph, how to assign local kernels to the enlarged local domains) make the procedure more of an art than a science.

---

[7] The whimsical term "moral graph" originally referred to the graph obtained from a DAG by drawing edges between – "marrying" – each of the parents of a given vertex [23].

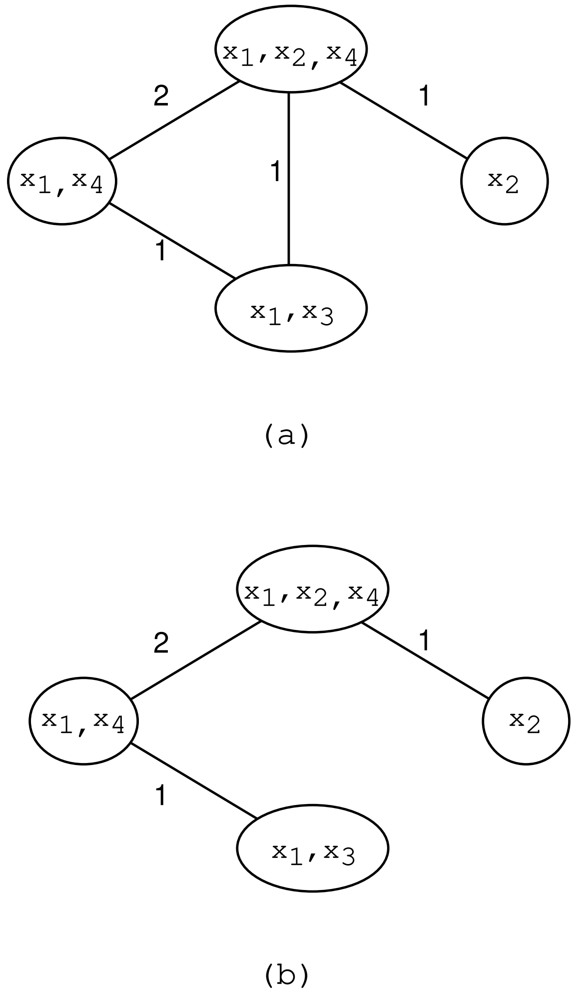

Figure 9 . (a) The local domain graph and (b) one junction tree for the local domains and kernels in Example 2.1.

For example, suppose the local domains and local kernels are

	local domain	local kernel
1.	$\{x_1, x_2\}$	$\alpha_1(x_1, x_2)$
2.	$\{x_2, x_3\}$	$\alpha_2(x_2, x_3)$
3.	$\{x_3, x_4\}$	$\alpha_3(x_3, x_4)$
4.	$\{x_1, x_4\}$	$\alpha_4(x_1, x_4)$

As we observed above, these local domains cannot be organized into a junction tree. The moral graph for these domains is shown in Figure 10 (a) (solid lines). This graph is not triangulated, but the addition of the edge 2–4 (dashed line) makes it so. The cliques in the triangulated graph are $\{1, 2, 4\}$ and $\{2, 3, 4\}$, and these sets can be made the labels in a junction tree (Figure 10 (b)). We can attach the original four local domains as leaves to this junction tree, as shown in Figure 10 (c) (note that this graph is identical to the junction tree in Figure 5). Finally, we can assign the local kernel at $\{1,4\}$ to the local domain $\{1,2,4\}$, and the local kernel at $\{3,4\}$ to the local domain $\{2,3,4\}$, thereby obtaining the junction tree shown in Figure 10 (d). What we have done, in effect, is to modify the original local domains by enlarging two of them, and viewing the associated local kernels as functions on the enlarged local domains:

	local domain	local kernel
1.	$\{x_1, x_2\}$	$\alpha_1(x_1, x_2)$
2.	$\{x_2, x_3\}$	$\alpha_2(x_2, x_3)$
$3'$.	$\{x_2, x_3, x_4\}$	$\alpha_3'(x_2, x_3, x_4) = \alpha_3(x_3, x_4)$
$4'$.	$\{x_1, x_2, x_4\}$	$\alpha_4'(x_1, x_2, x_4) = \alpha_4(x_1, x_4)$.

The difficulty is that we must enlarge the local domains enough so that they will support a junction tree, but not so much that the resulting algorithm will be unmanageably complex. We will return to the issue of junction tree complexity in Section 5.

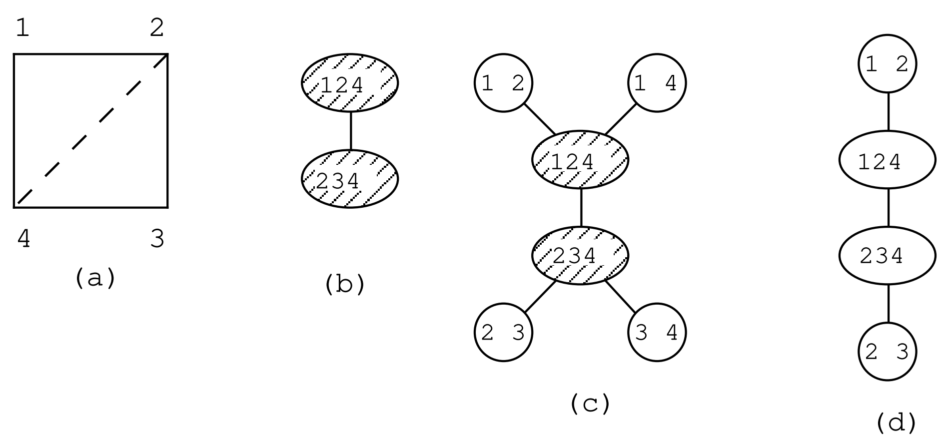

Figure 10 . Constructing a junction tree for the local domains $\{1,2\}$, $\{2,3\}$, $\{3,4\}$, and $\{4,1\}$ by triangulating the moral graph.

The next example illustrates this procedure in a more practical setting.

4.2 Example. Here we continue Example 2.2. The local domain graph is shown in Figure 11. Since all edges have weight 1, any spanning tree will have weight 4, but $w^* = 12 - 6 = 6$. Thus by Theorem 4.1, the local domains cannot be organized into a junction tree, so we need to consider the moral graph, which is shown in Figure 12 (a). It is not triangulated (e.g. the cycle formed by vertices y_1, y_2, x_2, and x_1 has no chord), but it can be triangulated by the addition of three additional edges, as shown in Figure 12 (b). There are exactly three cliques in the triangulated moral graph, viz., $\{y_1, y_2, y_3, x_1\}$, $\{y_2, y_3, x_1, x_2\}$, and $\{y_3, x_1, x_2, x_3\}$. These three sets can be organized into a unique junction tree, and each of the original 5 local domains is a subset of exactly one of these, as shown in Figure 13(a).

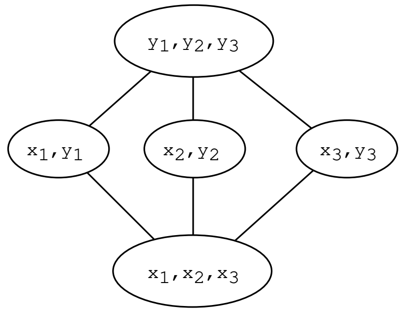

Figure 11 . The LD graph for the local domains and kernels in Example 2.2. (All edges have weight 1.) There is no junction tree.

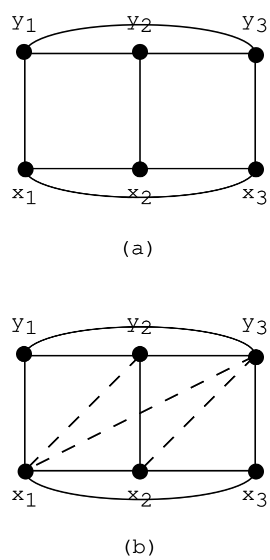

Figure 12 . The moral graph (top) and a triangulated moral graph (bottom) for the local domains and kernels in Example 2.2.

If we want a unique local domain for each of the 5 local kernels, we can retain two of the original local domains, thus obtaining the junction tree shown in Figure 13(b). Since this is a "single-vertex" problem, to apply the GDL, we first direct each of the edges towards the target vertex, which in this case is $\{x_1, x_2, x_3\}$. It is now a straightforward exercise to show that the (serial, one-vertex) GDL, when applied to this directed junction tree, yields the usual "fast" Hadamard transform. More generally, by extending the method in this example, it is possible to show that the FFT on any finite Abelian group, as described,

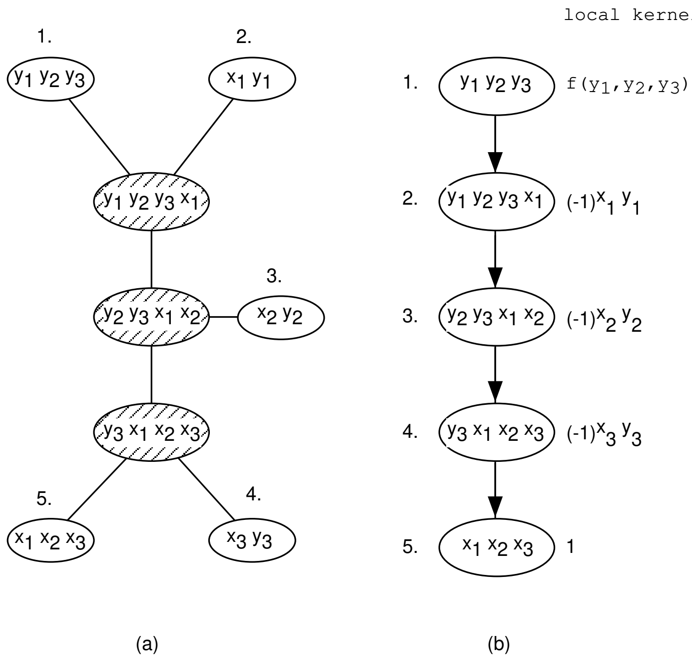

Figure 13 . Constructing a junction tree for Example 4.2.

e.g., in [8] or [31], can be derived from an application of the GDL.[8] ▪

4.3 Example. Here we continue Example 2.3. In this case the local domains can be organized as a junction tree. One such tree is shown in Figure 14. It can be shown that the GDL, when applied to the junction tree of Figure 14, yields the Gallager-Tanner-Wiberg algorithm [15][34][39] for decoding linear codes defined by cycle-free graphs. Indeed, Figure 14 is identical to the "Tanner graph" cited by Wiberg [39] for decoding this particular code. ▪

4.4 Example. Here we continue Example 2.4. The local domains can be arranged into a junction tree, as shown in Figure 15. (In general, the junction tree has the same topology as DAG, if the DAG is cycle-free.) The GDL algorithm, when applied to the junction tree of Figure 15, is equivalent to certain algorithms which are known in the artificial intelligence community for solving the probabilistic inference problem on Bayesian networks whose associated DAGs are cycle-free; in particular Pearl's "belief propagation" algorithm [29], and the "probability propagation" algorithm of Shafer and Shenoy [33]. ▪

4.5 Example. Here we continue Example 2.5, the probabilistic state machine. In this case the local domains can be organized into a junction tree, as illustrated in Figure 16

[8] For this, see [1], where it is observed that the moral graph for the DFT over a finite Abelian group G is triangulated if and only if G is a cyclic group of prime-power order. In all other cases, it is necessary to triangulate the moral graph, as we have done in this example.

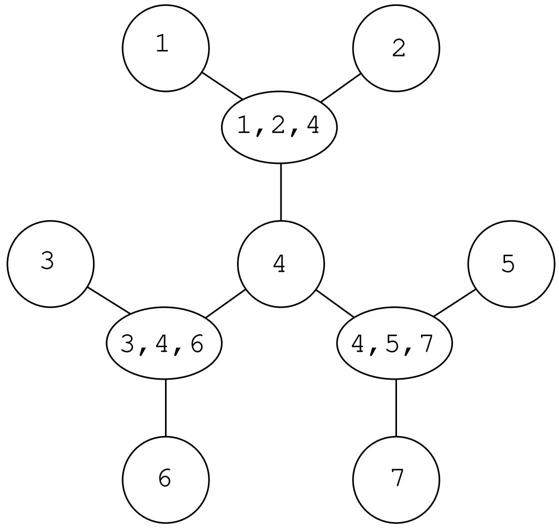

Figure 14 . A junction tree for Example 4.3.

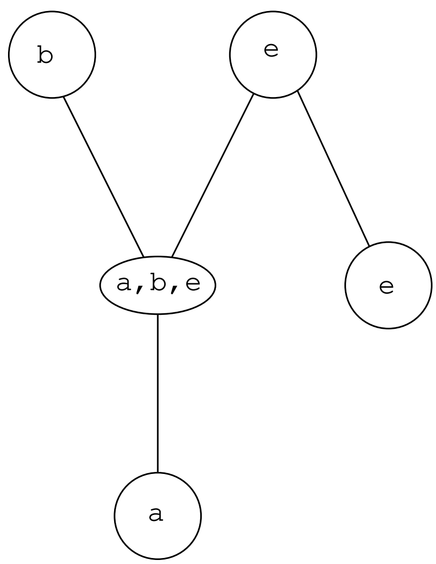

Figure 15 . A junction tree for Example 4.4.
This figure should be compared to Figure 1.

for the case $n = 4$. The GDL algorithm, applied to the junction tree of Figure 16, gives us essentially the BCJR [5] and Viterbi [37][11] algorithms respectively. (For Viterbi's algorithm, we take the negative logarithm of the objective function in eq. (2.5), and use the min-sum semiring, with a single target vertex, preferably the "last" $\{u_i\}$, which in Figure 16 is $\{u_3\}$. For the BCJR algorithm, we use the objective function in eq. (2.5) as it stands, and use the sum-product semiring, and evaluate the objective function at each of the vertices $\{u_i\}$, for $i = 0, \ldots, n-1$. In both cases the appropriate schedule is fully serial.) ▪

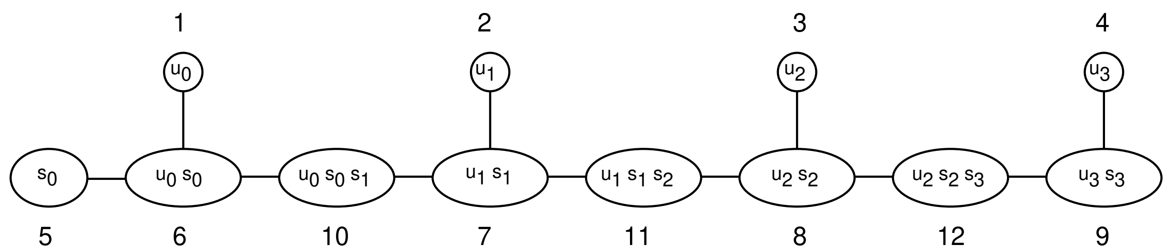

Figure 16 . A junction tree for the probabilistic state machine (illustrated for $n = 4$).

4.6 Example. Here we continue Example 2.6, the matrix multiplication problem. It is easy to see that for $n \geq 3$, there is no junction tree for the original set of local domains, because the corresponding moral graph is a cycle of length $n+1$. It is possible to show that for the product of n matrices, there are

$$\frac{1}{n}\binom{2n-2}{n-1}$$

possible triangulations of the moral graph, which are in one-to-one correspondence with the different ways to parenthesize the expression $M_1 \cdots M_n$. For example, the parenthesization

$$((\cdots(M_1M_2)\cdots)M_{n-1})M_n$$

corresponds to the triangulation shown in Figure 17.

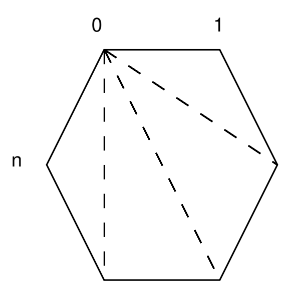

Figure 17 . The triangulation of the moral graph corresponding to the parenthesization $((\cdots(M_1M_2)\cdots)M_{n-1})M_n$.

Thus the problem of finding an optimal junction tree is identical to the problem of finding an optimal parenthesization. For example, in the case $n = 3$, illustrated in Figure 18, there are two different triangulations of the moral graph, which lead, via the techniques described in this section, to the two junction trees shown in the lower part of Figure 18. With the top vertex as the target, the GDL applied to each of these trees computes the product $M_1M_2M_3$. The left junction tree corresponds to parenthesizing the product $M_1M_2M_3$ as $(M_1M_2)M_3$ and requires $2q_0q_1q_2 + 2q_0q_2q_3$ arithmetic operations, whereas the right junction tree corresponds to $M_1(M_2M_3)$ and requires $2q_1q_2q_3 + 2q_0q_1q_3$ operations. Thus which tree one prefers depends on the relative size of the matrices. For example, if $q_0 = 10$, $q_1 = 100$, $q_2 = 5$, and $q_3 = 50$, the left junction tree requires 15000 operations and the right junction tree takes $150,000$. (This example is taken from [9].)

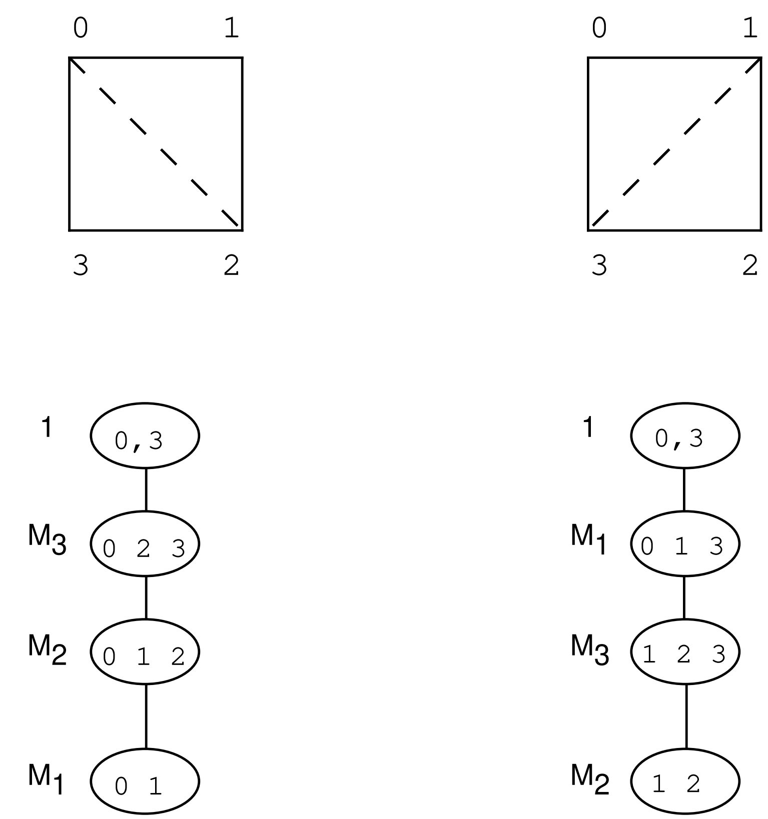

Figure 18 . The moral graph for Example 4.6, triangulated in two ways, and the corresponding junction trees. The left junction tree corresponds to the parenthesization $(M_1M_2)M_3$, and the one on the right corresponds to $M_1(M_2M_3)$.

As we discussed in Example 2.6, the matrix multiplication problem is equivalent to a trellis path problem. In particular, if the computations are in the min-sum semiring, the problem is that of finding the shortest paths in the trellis. If the moral graph is triangulated as shown in Figure 17, the resulting junction tree yields an algorithm identical to Viterbi's algorithm. Thus Viterbi's algorithm can be viewed as an algorithm for multiplying a chain of matrices in the min-sum semiring. (This connection is explored in more detail in [4].■

5. Complexity of the GDL.

In this section we will provide complexity estimates for the serial versions of the GDL discussed in Section 3. Here by complexity we mean the *arithmetic complexity*, i.e., the total number of (semiring) additions and/or multiplications required to compute the desired objective functions.

We begin by rewriting the message and state computation formulas (3.1) and (3.2), using slightly different notation. The message from vertex v to vertex w is defined as (cf. (3.1))

$$\mu_{v,w}(x_{v\cap w}) = \sum_{x_{v\setminus w}\in A_{S(v)\setminus S(w)}} \alpha_v(x_v) \prod_{\substack{u \text{ adj } v \\ u\neq v}} \mu_{u,v}(x_{u\cap v}). \tag{5.1}$$

and the state of vertex v is defined as (cf. (3.2))

$$\sigma_v(x_v) = \alpha_v(x_v) \prod_{u \text{ adj } v} \mu_{u,v}(x_{u\cap v}). \tag{5.2}$$

We first consider the single vertex problem, supposing that v_0 is the target. For each $v \neq v_0$, there is exactly one edge directed from v toward v_0. We suppose that this edge is (v, w). To compute the message $\mu_{v,w}(x_{v\cap w})$ as defined in (5.1) for a particular value of $x_{v\cap w}$ requires[9] $|A_{S(v)\setminus S(w)}|-1$ additions, and $|A_{S(v)\setminus S(w)}|(d(v)-1)$ multiplications, where $d(v)$ is the degree of the vertex v. Using simplified but (we hope) self-explanatory notation we rewrite this as follows:

$$\begin{aligned} &q_{v\setminus w} - 1 \text{ additions, and} \\ &q_{v\setminus w}\cdot(d(v)-1) \text{ multiplications.} \end{aligned}$$

But there are $q_{v\cap w} \stackrel{\text{def}}{=} |A_{S(v)\cap S(w)}|$ possibilities for $x_{v\cap w}$, so the entire message $\mu_{v,w}(x_{v\cap w})$ requires

$$\begin{aligned} &(q_{v\cap w})(q_{v\setminus w} - 1) = q_v - q_{v\cap w} \text{ additions, and} \\ &(q_{v\cap w})q_{v\setminus w}\cdot(d(v)-1) = (d(v)-1)q_v \text{ multiplications.} \end{aligned}$$

The total number of arithmetic operations required to send messages toward v_0 along each of the edges of the tree is thus

$$\begin{aligned} &\sum_{v\neq v_0}(q_v - q_{v\cap w}) \qquad \text{additions} \\ &\sum_{v\neq v_0}(d(v)-1)q_v \qquad \text{multiplications.} \end{aligned}$$

[9] Here we are assuming that the addition (multiplication) of N elements of S requires $N-1$ binary additions (multiplications).

When all the messages have been computed and transmitted, the algorithm terminates with the computation of the state at v_0, defined by (5.2). This state computation requires $d(v_0)q_{v_0}$ further multiplications, so that the total is

$$\sum_{v\neq v_0} (q_v - q_{v\cap w}) \qquad \text{additions}$$
$$\sum_{v\neq v_0} (d(v)-1)q_v + d(v_0)q_{v_0} \qquad \text{multiplications.}$$

Thus the grand total number of additions and multiplications is

$$\chi(T) = \sum_{v\in V} d(v)q_v - \sum_{e\in E} q_e, \tag{5.3}$$

where if $e = (v, w)$ is an edge, its "size" q_e is defined to be $q_{v\cap w}$.

Note that the formula in (5.3) gives the upper bound

$$\chi(T) \leq \sum_{v\in V} d(v)q_v \tag{5.4}$$

mentioned in Section 3.

The formula in (5.3) can be rewritten in a useful alternative way, if we define the "complexity" of the edge $e = (v, w)$ as

$$\chi(e) = q_v + q_w - q_{v\cap w}. \tag{5.5}$$

With this definition, the formula in (5.3) becomes

$$\chi(T) = \sum_{e\in E} \chi(e). \tag{5.6}$$

For example, for the junction tree of Figure 4, there are four edges and

$$\begin{aligned}
\chi(v_1, v_2) &= q_1 + q_1q_2 - q_1 = q_1q_2\\
\chi(v_1, v_3) &= q_1 + q_1q_3 - q_1 = q_1q_3\\
\chi(v_2, v_4) &= q_1q_2 + q_2 - q_2 = q_1q_2\\
\chi(v_2, v_5) &= q_1q_2 + q_2q_4 - q_2,
\end{aligned}$$

so that $\chi(T) = 3q_1q_2 + q_1q_3 + q_2q_4 - q_2$.

We next briefly consider the all-vertices problem. Here a message must be sent over each edge, in both directions, and the state must be computed at each vertex. If this is done following the ideas above in the obvious way, the resulting complexity is $O(\sum_v d(v)^2 q_v)$. However, we may reduce this by noticing that if $\{a_1, \ldots, a_d\}$ is a set of d numbers, it is possible to compute all the d products of $d-1$ of the a_i's with at most $3(d-2)$

multiplications, rather than the obvious $d(d-2)$. We do this by precomputing the quantities $b_1 = a_1$, $b_2 = b_1 \cdot a_2 = a_1 a_2$, $\ldots$, $b_{d-1} = b_{d-2} \cdot a_{d-1} = a_1 a_2 \cdots a_{d-1}$, and $c_d = a_d$, $c_{d-1} = a_{d-1} c_d = a_{d-1} a_d$, $\ldots$, $c_2 = a_2 \cdot c_3 = a_2 a_3 \cdots a_d$, using $2(d-2)$ multiplications. Then if $\widehat{a}_j$ denotes the product of all the a_i's except for a_j, we have $\widehat{a}_1 = c_2$, $\widehat{a}_2 = b_1 \cdot c_3$, $\ldots$, $\widehat{a}_{d-1} = b_{d-2} \cdot c_d$, $\widehat{a}_d = b_{d-1}$, using a further $d-2$ multiplications, for a total of $3(d-2)$. With one further multiplication ($b_{d-1} \cdot a_d$), we can compute $b_d = a_1 a_2 \cdots a_d$.[10]

Returning now to the serial implementation of the all-vertex GDL, each vertex must pass a message to each of its neighbors. Vertex v will have $d(v)$ incoming messages, and (prior to marginalization) each outgoing message will be the product of $d(v) - 1$ of these messages with the local kernel at v. For its own state computation, v also needs the product of all $d(v)$ incoming messages with the local kernel. By the above argument, all this can be done with at most $3d(v)$ multiplications for each of the q_v values of the variables in the local domain at v. Thus the number of multiplications required is at most $3\sum_v d(v) q_v$. The marginalizations during the message computations remain as in the single-vertex case, and, summed over all messages, require

$$\sum_{v \in V} d(v) q_v - 2 \sum_{e \in E} q_e$$

additions. Thus the total number of arithmetic operations is no more than $4\sum_v d(v) q_v$, which shows that the complexity of the all-vertices GDL is at worst a fixed constant times that of the single-vertex GDL. Therefore we feel justified in *defining* the complexity of a junction tree, irrespective of which objective functions are sought, by (5.3) or (5.6). (In [23], the complexity of a similar, but not identical, algorithm was shown to be upperbounded by $3\sum_v q_v + M \max_v q_v$. This bound is strictly greater than the bound in (5.4).)

In Section 4, we saw that in many cases $w_{\max} = w^*$ and the LD graph has more than one maximal-weight spanning tree. In view of the results in this section, in such cases it is desirable to find the maximal-weight spanning tree with $\chi(T)$ as small as possible. It is easy to modify Prim's algorithm to do this. In Prim's algorithm, the basic step is to add to the growing tree a maximal-weight edge which does not form a cycle. If there are several choices with the same weight, choose one whose complexity, as defined by (5.5), is as small as possible. The tree that results is guaranteed to be a minimum-complexity junction tree [19]. In fact, we used this technique to find minimum-complexity junction trees in Examples 4.1, 4.3, 4.4, and 4.5.

We conclude this section with two examples which illustrate the difficulty of finding the minimum-complexity junction tree for a given marginalization problem. Consider first the local domains $\{x_1\}$, $\{x_2\}$, $\{x_3\}$, and $\{x_1, x_2, x_3\}$. There is a unique junction tree with these sets as vertex labels, shown in Figure 19 (a). By (5.3), the complexity of this junction tree is $3q_1q_2q_3$. Now suppose we artificially enlarge the local domain $\{x_2\}$ to $\{x_1, x_2\}$. Then the

[10] One of the referees has noted that the trick described in this paragraph is itself an application of the GDL; it has the same structure as the forward-backward algorithm applied to a trellis representing a repetition code of length d.

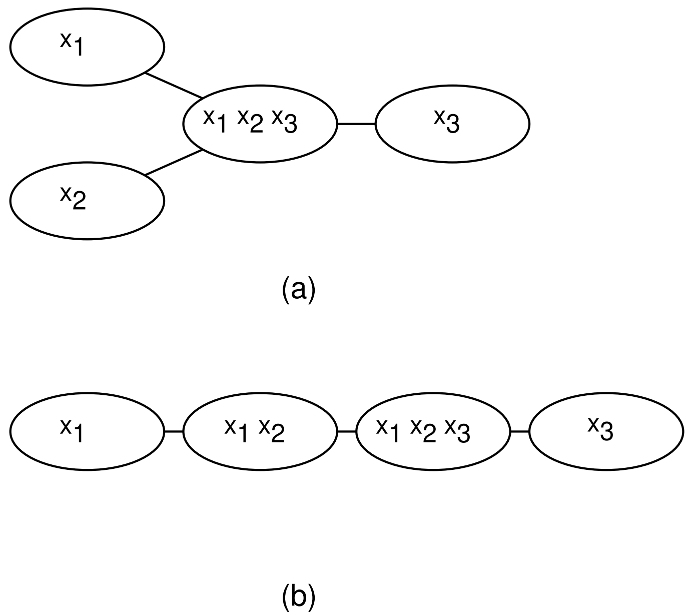

Figure 19 . Enlarging a local domain can lower the junction tree complexity.

modified set of local domains, viz., $\{x_1\}$, $\{x_1, x_2\}$, $\{x_3\}$, and $\{x_1, x_2, x_3\}$ can be organized into the junction tree shown in Figure 19 (b), whose complexity is $2q_1q_2q_3 + q_1q_2$, which is less than that of the original tree as long as $q_3 > 1$.

As the second example, we consider the domains $\{x_1, x_4\}$, $\{x_2, x_5\}$, $\{x_3, x_6\}$, and $\{x_1, x_2, x_3\}$, which can be organized into a unique junction tree (Figure 20(a)). If we adjoin the domain $\{x_2, x_3, x_5\}$, however, we can build a junction tree (Figure 20(b)) whose complexity is lower than the original one, provided that q_1 is much larger than any of the other q_i's. (It is known that the problem of finding the "best" triangulation of a given graph is NP-complete [40], where "best" refers to having the minimum maximum clique size.)

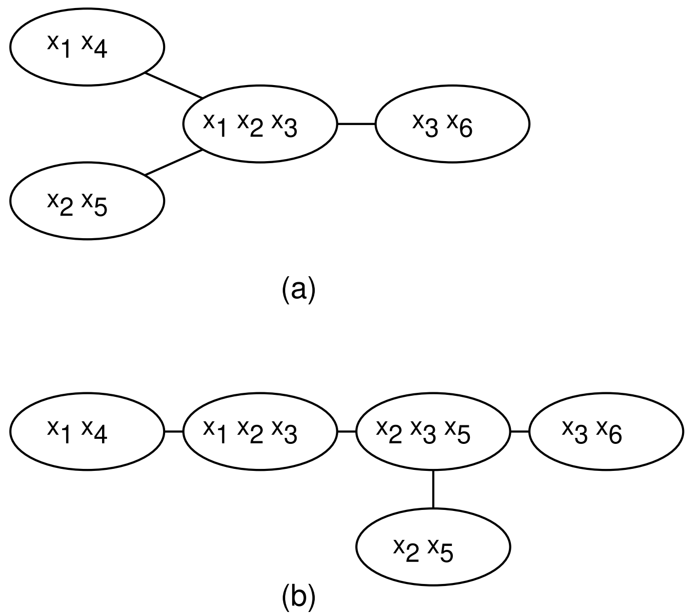

Figure 20 . Adding an extra local domain can lower the junction tree complexity.

6. A Brief History of the GDL.

Important algorithms whose essential underlying idea is the exploitation of the distributive law to simplify a marginalization problem have been discovered many times in the past. Most of these algorithms fall into one of three broad categories: *decoding algorithms*, the *"forward-backward algorithm,"* and *artificial intelligence algorithms.* In this section we will summarize these three parallel threads.

• Decoding Algorithms.

The earliest occurrence of a GDL-like algorithm that we are aware of is Gallager's 1962 algorithm for decoding low-density parity-check codes [15][16]. Gallager was aware that his algorithm could be proved to be correct only when the underlying graphical structure has no cycles, but also noted that it gave good experimental results even when cycles were present. Gallager's work attracted little attention for 20 years, but in 1981 Tanner [34], realizing the importance of Gallager's work, made an important generalization of low-density parity check codes, introduced the "Tanner graph" viewpoint, and recast Gallager's algorithm in explicit message-passing form. Tanner's work itself went relatively unnoticed until the 1996 thesis of Wiberg [39], which showed that the message-passing Tanner graph decoding algorithm could be used not only to describe Gallager's algorithm, but also Viterbi's and BCJR's. Wiberg too understood the importance of the cycle-free condition, but nevertheless observed that the turbo-decoding algorithm was an instance of the Gallager-Tanner-Wiberg algorithm on a graphical structure with cycles. Wiberg explicitly considered both the sum-product and min-sum semirings, and speculated on the possibility of further generalizations to what he called "universal algebras" (our semirings).

In an independent series of developments, in 1967 Viterbi [37] invented his celebrated algorithm for maximum-likelihood decoding (minimizing sequence error probability) of convolutional codes. Seven years later (1974), Bahl, Cocke, Jelinek, and Raviv [5] published a "forward-backward" decoding algorithm (see next bullet) for minimizing the bit error probability of convolutional codes. The close relationship between these two algorithms was immediately recognized by Forney [11]. Although these algorithms did not apparently lead anyone to discover a class of algorithms of GDL-like generality, with hindsight we can see that all the essential ideas were present.

• The forward-backward algorithm.

The forward-backward algorithm (also known as the M-step in the Baum-Welch algorithm) was invented in 1962 by Lloyd Welch, and seems to have first appeared in the unclassified literature in two independent 1966 publications [6][7]. It appeared explicitly as an algorithm for tracking the states of a Markov chain in the early 1970's [26][5] (see also the survey articles [30] and [32]). A similar algorithm (in min-sum form) appeared in a 1971 paper on equalization [35]. The algorithm was connected to the optimization literature in 1987 [36], where a semiring-type generalization was given.

• Artificial Intelligence.

The relevant research in the AI community began relatively late, but it has evolved quickly. The activity began in the 1980's with the work of Kim and Pearl [20] and Pearl [29]. Pearl's "belief propagation" algorithm, as it has come to be known, is a message-passing algorithm for solving the probabilistic inference problem on a Bayesian network whose DAG contains no (undirected) cycles. Soon afterwards, Lauritzen and Spiegelhalter [23] obtained an equivalent algorithm, and moreover generalized it to arbitrary DAGs by introducing the triangulation procedure. The notion of junction trees (under the name "Markov tree") was explicitly introduced by Shafer and Shenoy [33]. A recent book by Jensen [18] is a good introduction to most of this material. A recent unification of many of these concepts called "bucket elimination" appears in [10], and a recent paper by Lauritzen and Jensen [22] abstracts the MPF problem still further, so that the marginalization is done axiomatically, rather than by summation.

In any case, by early 1996, the relevance of these AI algorithms had become apparent to researchers in the information theory community [28][21]. Conversely, the AI community has become excited by the developments in the information theory community [38][14], which demonstrate that these algorithms can be successful on graphs with cycles. We discuss this is in the next section.

7. Iterative and Approximate Versions of the GDL.

Although the GDL can be proved to be correct only when the local domains can be organized into a junction tree, the computations of the messages and states in (3.1) and (3.2) make sense whenever the local domains are organized as vertex labels on any kind of a connected graph, whether it is a junction tree or not. On such a junction graph, there is no notion of "termination," since messages may travel around the cycles indefinitely. Instead, one hopes that after sufficiently many messages have been passed, the states of the selected vertices will be *approximately equal* to the desired objective functions. This hope is based on a large body of experimental evidence, and some emerging theory.

• Experimental Evidence.

It is now known that an application of the GDL, or one of its close relatives, to an appropriate junction graph with cycles, gives both the Gallager-Tanner-Wiberg algorithm for low-density parity-check codes [24][25][39][28], and the turbo decoding algorithm [39][28][21]. Both of these decoding algorithms have proved to be extraordinarily effective experimentally, despite the fact that there are as yet no general theorems that explain their behavior.

• Emerging Theory—Single-Cycle Junction Graphs..

Recently, a number of authors [39][38][2][12][3][1] have studied the behavior of the iterative GDL on junction graphs which have exactly one cycle. It seems fair to say that, at least for the sum-product and the min-sum semirings, the iterative GDL is fairly well understood in this case, and the results imply, for example, that iterative decoding is effective for most tail-biting codes. Although these results shed no direct light on the problem of the behavior of the GDL on multi-cycle junction graphs, like those associated with Gallager codes or turbo codes, this is nevertheless an encouraging step.

Appendix A. Proof of the Scheduling Theorem.

Summary: In this appendix, we will give a proof of the Scheduling Theorem 3.1, which will prove the correctness of the GDL. The key to the proof is Corollary A.4, which tells us that at every stage of the algorithm, the state at a given vertex is the appropriately marginalized product of a subset of the local kernels. Informally, we say that at time t, the state at vertex v is the marginalized product of the local kernels which are currently "known" to v. Given this result, the remaining problem is to understand how knowledge of the local kernels is disseminated to the vertices of the junction tree under a given schedule. As we shall see, this "knowledge dissemination" can be described recursively as follows:

- *Rule (1): Initially ($t = 0$), each vertex v_i knows only its own local kernel α_i.*

- *Rule (2): If a directed edge (v_i, v_j) is activated at time t, i.e., if $(v_i, v_j) \in E_t$, then vertex v_j learns all the local kernels known to v_i at time $t - 1$.*

The proof of Theorem 3.1 then follows quickly from these rules.

We begin by introducing some notation. Let $f(x_S)$ be a function of the variable list x_S, and let T be an arbitrary subset of $\{1, \ldots, n\}$. We denote by $[f(x_S)]^T$ the function of the variable list $x_{S \cap T}$ obtained by "marginalizing out" the variables in f which are not in T:

$$[f(x_S)]^T \stackrel{\text{def}}{=} \sum_{x_{S \setminus T}} f(x_S).$$

A.1 Lemma. *If $S \cap U \subseteq T$, then*

$$[[f(x_S)]^T]^U = [f(x_S)]^U.$$

Proof: (Note first that Lemma A.1 is a special case of the single-vertex GDL, with the following local domains and kernels.

	local domain	local kernel
1.	$\{x_S\}$	$f(x_S)$
2.	$\{x_T\}$	1
3.	$\{x_U\}$	1.

The appropriate junction tree is shown in Figure 21.)

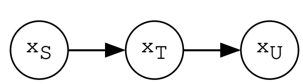

Figure 21 . A junction tree for Lemma A.1.

To see that the assertion is true, note that the variables not marginalized out in the function $[[f(x_S)]^T]^U$ are those indexed by $S \cap T \cap U$. The variables not marginalized out in $[f(x_S)]^U$ are those indexed by $S \cap U$. But by the hypothesis $S \cap U \subseteq T$, these two sets are equal. ▪

A.2 Lemma. *Let $f_i(x_{S_i})$, for $i = 1, \ldots, K$ be local kernels, and consider the MPF problem of computing*

$$\beta(x_{(S_1 \cup \cdots S_K) \cap T}) \stackrel{\text{def}}{=} [\prod_{i=1}^{K} f_i(x_{S_i})]^T. \tag{A.1}$$

If no variable which is marginalized out in (A.1) occurs in more than one local kernel, i.e., if $S_i \cap S_j \subseteq T$ for $i \neq j$, then

$$\beta(x_{(S_1 \cup \cdots S_K) \cap T}) = \prod_{i=1}^{K} [f_i(x_{S_i})]^T .$$

Proof: (Lemma A.2 is also a special case of the single-vertex GDL, with the following local domains and kernels.

	local domain	local kernel
1	$\{x_{S_1}\}$	$f_1(x_{S_1})$
$\vdots$	$\vdots$	$\vdots$
K	$\{x_{S_K}\}$	$f_K(x_{S_K})$
$K+1$	$\{x_T\}$	1.

The appropriate junction tree is shown in Figure 22.)

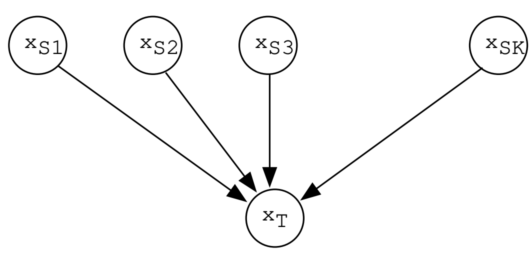

Figure 22 . A junction tree for Lemma A.2.

In any case, Lemma A.2 is a simple consequence of the distributive law: Since each variable being marginalized out in (A.1) occurs in at most one local kernel, it is allowable to take the other local kernels out of the sum by distributivity. As an example, we have

$$\sum_{x_1, x_3, x_4} f_1(x_1, x_2) f_2(x_2, x_3) f_3(x_2, x_4) = \sum_{x_1} f_1(x_1, x_2) \sum_{x_3} f_2(x_2, x_3) \sum_{x_4} f_3(x_2, x_4).$$

$\blacksquare$

Now we are ready to consider the dynamics of the GDL. Consider an edge: $e_{ij} : v_i \to v_j$. Removing e_{ij} from the junction tree T breaks it into two components, T_{ij} and T_{ji} (see Figure 23). For future reference, we denote the vertex set of T_{ij} by V_{ij}, and the edge set by E_{ij}

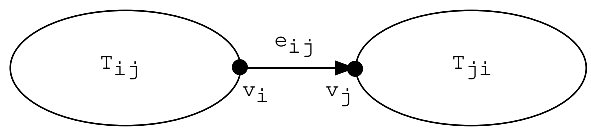

Figure 23 . Deleting the edge e_{ij} breaks the junction tree into two components.

Since e_{ij} is on the unique path between any vertex in T_{ij} and any vertex in T_{ji}, it follows from the junction tree property that any variable which occurs in a vertex in both components must occur in both v_i and v_j. Thus the message $\mu_{i,j}$, which may be viewed as a message from T_{ij} to T_{ji}, is a function of exactly those variables which occur in both components.

In what follows, for each index $i = 1, \ldots, M$ we define

$$N_i = \{k : (v_k, v_i) \in E\},$$

and for each pair of indices (i, j) such that $(v_i, v_j) \in E$, we define

$$N_{i,j} = \{k : (v_k, v_i) \in E, k \neq j\}.$$

In words, N_i represents the (indices of) the neighbors of v_i, and $N_{i,j}$ represents the (indices of) the neighbors of v_i other than v_j.

Now let $\mathcal{E} = (E_1, E_2, \ldots, E_N)$ be a schedule for a junction tree, as defined in Section 3, i.e., a finite list of subsets of E, and let $\mu_{i,j}(t)$ be the value of the message $\mu_{i,j}$ after the tth round of $\mathcal{E}$.

A.3 Theorem. *The message $\mu_{i,j}(t)$ is the product of a subset of the local kernels in T_{ij}, with the variables that do not occur in T_{ji} marginalized out. Specifically, we have*

$$\mu_{i,j}(t) = \left[\prod_{k \in K_{i,j}(t)} \alpha_k \right]^{S_j}, \tag{A.2}$$

where $K_{i,j}(t)$ is a subset of V_{ij}, the vertex set of T_{ij}. The sets $K_{i,j}(t)$ are defined inductively as follows:

$$\begin{aligned} &K_{i,j}(0) = \emptyset, \qquad \textit{and for } t \geq 1, \\ &K_{i,j}(t) = \begin{cases} K_{i,j}(t-1) & \textit{if } e_{ij} \notin E_t \\ \{i\} \bigcup_{\ell \in N_{i,j}} K_{\ell,i}(t-1) & \textit{if } e_{ij} \in E_t. \end{cases} \end{aligned} \tag{A.3}$$

Proof: We use induction on t, the case $t = 0$ being simply a restatement of the initialization rule $\mu_{i,j} = 1$. Assuming the theorem proved for $t-1$, we assume $e_{i,j}$ is updated in the tth round, and consider $\mu_{i,j}(t)$:

$$\begin{aligned}
\mu_{i,j}(t) &= \left[\alpha_i \prod_{\ell \in N_{i,j}} \mu_{\ell,i}(t-1)\right]^{S_j} && \text{by (3.1)} \\
&= \left[\alpha_i \prod_{\ell \in N_{i,j}} \left[\prod_{k \in K_{\ell,i}(t-1)} \alpha_k\right]^{S_i}\right]^{S_j} && \text{by induction.}
\end{aligned}$$

Any variable that occurs in two different messages $\mu_{\ell_1,i}(t-1)$ and $\mu_{\ell_2,i}(t-1)$ must also, by the junction tree property, occur in α_i, so we may apply Lemma A.2 to rewrite the last line as

$$= \left[\left[\alpha_i \prod_{\ell \in N_{i,j}} \prod_{k \in K_{\ell,i}(t-1)} \alpha_k\right]^{S_i}\right]^{S_j}.$$

Since a variable that occurs in one of the kernels in the above equation and also in v_j must, by the junction tree property, also occur in v_i, it follows from Lemma A.1 that this last expression can be simplified to

$$\left[\alpha_i \prod_{\ell \in N_{i,j}} \prod_{k \in K_{\ell,i}(t-1)} \alpha_k\right]^{S_j} = \left[\prod_{k \in K_{i,j}(t)} \alpha_k\right]^{S_j},$$

the last equality because of the definition (A.3). ▪

A.4 Corollary. *For all $t \geq 0$, the state $\sigma_i(t)$ has the value*

$$\sigma_i(t) = \left[\prod_{k \in J_i(t)} \alpha_k\right]^{S_i}, \tag{A.4}$$

where the set $J_i(t)$ is defined by

$$J_i(t) = \{i\} \bigcup_{j \in N_i} K_{j,i}(t). \tag{A.5}$$

Proof: By definition (3.2),

$$\begin{aligned}
\sigma_i(t) &= \alpha_i \prod_{j \in N_i} \mu_{j,i}(t) \\
&= \alpha_i^{S_i} \prod_{j \in N_i} \mu_{j,i}(t).
\end{aligned}$$

(We know that $\alpha_i = \alpha_i^{S_i}$, since the kernel α_i is by definition a function only of the variables involved in the local domain S_i.) By Theorem A.3, this can be written as

$$\sigma_i(t) = \alpha_i^{S_i} \prod_{j \in N_i} \left[\prod_{k \in K_{j,i}(t)} \alpha_k \right]^{S_i}.$$

But by the junction tree property, any variable that occurs in two of the bracketed terms must also occur in α_i, so that by Lemma A.2,

$$\begin{aligned} \sigma_i(t) &= \left[\alpha_i \prod_{j \in N_i} \prod_{k \in K_{j,i}(t)} \alpha_k \right]^{S_i} \\ &= \left[\prod_{k \in J_i(t)} \alpha_k \right]^{S_i} \qquad \text{by the definition (A.5).} \quad ▪ \end{aligned}$$

Theorem A.3 tells us that at time t, the message from v_i to v_j is the appropriately marginalized product of a subset of the local kernels, viz., $\{\alpha_k : k \in K_{i,j}(t)\}$, and Corollary A.4 tells us that at time t, the state of vertex v_i is the appropriately marginalized product of a subset of the local kernels, viz., $\{\alpha_k : k \in J_i(t)\}$, which we think of as the subset of local kernels which are "known" to v_i at time t. Given these results, the remaining problem is to understand how knowledge of the local kernels is disseminated to the vertices of the junction tree under a given schedule. A study of equation (A.5), which gives the relationship between what is known at the vertex v_i and what is known by the incoming edges, together with the message update rules in (A.3), provides a a nice recursive description of exactly how this information is disseminated:

• Rule (1): Initially ($t = 0$), each vertex v_i knows only its own local kernel α_i.

• Rule (2): If a directed edge (v_i, v_j) is activated at time t, i.e., if $(v_i, v_j) \in E_t$, then vertex v_j learns all the local kernels previously known to v_i at time $t - 1$.

We shall now use these rules to prove Theorem 3.1.

Theorem 3.1 asserts that v_i knows each of the M local kernels $\alpha_1, \ldots, \alpha_M$ at time $t = N$ if and only if there is a path in the message trellis from $v_j(0)$ to $v_i(N)$, for all $j = 1, 2, \ldots, M$. We will now prove the slightly more general statement that v_i knows α_j at time $t = N$ if and only if there is a path in the message trellis from $v_j(0)$ to $v_i(N)$.

To this end, let us first show that if v_i knows α_j at $t = N$, then there must be a path in the message trellis from $v_j(0)$ to $v_i(N)$. Because we are in a tree, there is a unique path from v_j to v_i, say

$$\mathbb{P}_{j,i} = [v_{k_0}, v_{k_1}, \ldots, v_{k_{L-1}}, v_{k_L}],$$

where $k_0 = j$ and $k_L = i$. Denote by t_ℓ the first (smallest) time index for which v_{k_ℓ} knows α_j. Then by Rule 2 and an easy induction argument, we have

$$(A.6) \qquad 1 \le t_1 < t_2 < \cdots < t_L \le N.$$

(In words, knowledge of α_j must pass sequentially to v_i through the vertices of the path $\mathbb{P}_{j,i}$.) In view of (A.6), we have the following path from $v_j(0) = v_{k_0}(0)$ to $v_i(N) = v_{k_L}(N)$ in the message trellis from $v_j(0)$ to $v_i(N)$:

$$(A.7) \qquad [v_{k_0}(0), \ldots, v_{k_0}(t_1 - 1); v_{k_1}(t_1), \ldots, v_{k_1}(t_2 - 1); \ldots; v_{k_L}(t_L), \ldots, v_{k_L}(N)].$$

Conversely, suppose there is a path from $v_j(0)$ to $v_i(N)$ in the message trellis. Then since apart from "pauses" at a given vertex, this path in the message trellis must be the unique path $\mathbb{P}_{i,j}$ from v_j to v_i, Rule 2 implies that knowledge of the kernel α_j sequentially passes through the vertices on the path $\mathbb{P}_{i,j}$, finally reaching v_i at time N.

This completes the proof of Theorem 3.1.

References.

1. S. M. Aji, *Graphical Models and Iterative Decoding.* Ph.D. thesis, Caltech, 1999.
2. S. M. Aji, G. B. Horn, and R. J. McEliece, "On the convergence of iterative decoding on graphs with a single cycle," Proc. CISS 98 (Princeton, March 1998).
3. S. M. Aji, G. B. Horn, R. J. McEliece, and M. Xu, "Iterative min-sum decoding of tail-biting codes." Proc. IEEE Information Theory Workshop, Killarney Ireland, June 1998. pp. 68–69.
4. S. M. Aji, R. J. McEliece, and M. Xu, "Viterbi's algorithm and matrix multiplication." Presented at CISS 1999, March 1999.
5. L. R. Bahl, J. Cocke, F. Jelinek, J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20 (March 1974), pp. 284–287.
6. L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite-state Markov chains," *Ann. Math. Stat.*, vol 37 (1966), pp. 1559–1563.
7. R. W. Chang and J. C. Hancock, "On receiver structures for channels having memory," *IEEE Trans. Inform. Theory*, vol. IT-12 (Oct. 1966), pp. 463–468.
8. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19 (April 1965), p. 297.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, Mass.: MIT-McGraw-Hill, 1990.
10. R. Dechter, "Bucket elimination: A unifying framework for probabilistic inference." To appear in *Learning and Inference in Graphical Models*, 1998.
11. G. D. Forney, Jr., "The Viterbi algorithm," *Proc. IEEE*, vol. 61 (March 1973), pp. 268–278.
12. G. D. Forney,Jr., F. R. Kschischang, and B. Marcus, "Iterative decoding of tail-biting trellises." Presented at IEEE Information Theory Workshop, San Diego, California, February 1998.
13. B. J. Frey, *Bayesian Networks for Pattern Classification, Data Compression, and Channel Coding.* University of Toronto Ph.D. thesis, 1997.
14. B. J. Frey and D. J. C. MacKay, "A revolution: Belief propagation in graphs with cycles," Proc. 1997 Neural Information Processing Systems Conference, in press.
15. R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8 (Jan. 1962), pp. 21–28.
16. R. G. Gallager, *Low-Density Parity-Check Codes.* Cambridge, Mass.: MIT Press, 1963.
17. R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Reading, Mass.: Addison, Wesley, 1992.

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

19. F. V. Jensen and F. Jensen, "Optimal junction trees," pp. 360–366 in *Proc. 10th Conf. on Uncertainty in Artificial Intelligence*, R. L. de Mantaras and D. Poole, eds. San Francisco: Morgan Kaufmann, 1994.

20. J. H. Kim and J. Pearl, "A computational model for causal and diagnostic reasoning," *Proc. 8th International Joint Conf. Artificial Intelligence* (1983), pp. 190–193.

21. F. R. Kschischang and B. J. Frey, "Iterative decoding of compound codes by probability propagation in graphical models." *IEEE J. Sel. Areas Comm.* vol. 16, no. 2 (Feb. 1998), pp. 219–230.

22. S. L. Lauritzen and F. V. Jensen, "Local computation with valuations from a commutative semigroup," *Annals Math. AI*, vol. 21, no. 1 (1997), pp. 51–69.

23. S. L. Lauritzen and D. J. Spiegelhalter, "Local computation with probabilities on graphical structures and their application to expert systems," *J. R. Statist. Soc.* B (1988), pp. 157–224.

24. D. J. C. MacKay and R. M. Neal, "Good codes based on very sparse matrices," pp. 100-111 in *Cryptography and Coding*, 5th IMA conference, Springer Lecture notes in Computer Science no. 1025. Berlin: Springer-Verlag, 1995.

25. D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity-check codes," *Electronics Letters*, vol 32 (1996), pp. 1645–1646. Reprinted in *Electronics Letters*, vol 33 (1996), pp. 457–458.

26. P. L. McAdam, L. R. Welch, and C. L. Weber, "M.A.P. bit decoding of convolutional codes," Proc. *1972 IEEE International Symposium on Information Theory*, (Asilomar, California, Jan. 1972) p. 91.

27. R. J. McEliece, R. B. Ash, and C. Ash. *Introduction to Discrete Mathematics.* New York: Random House, 1989.

28. R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm." *IEEE J. Sel. Areas Comm.* vol. 16, no. 2 (Feb. 1998), pp. 140–152.

29. J. Pearl, *Probabilistic Reasoning in Intelligent Systems.* San Mateo, CA: Morgan Kaufmann, 1988.

30. A. M. Poritz, "Hidden Markov models: a guided tour," *Proc. 1988 IEEE Conf. Acoustics, Speech, and Signal Processing*, vol. 1, pp. 7–13.

31. E. C. Posner, "Combinatorial structures in planetary reconnaissance," in *Error Correcting Codes*, H. B. Mann, ed. New York: John Wiley and Sons, 1968.

32. L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. IEEE*, vol. 77 (1989), pp. 257–285.

33. G. R. Shafer and P. P. Shenoy, "Probability propagation," *Ann. Math. Art. Intel.*,

vol. 2 (1990), pp. 327–352.

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

35. G. Ungerboeck, "Nonlinear equalization of binary signals in Gaussian noise," *IEEE Trans. Comm. Tech.*, vol. COM-19 (Dec. 1971), pp. 1128–1137.

36. S. Verdu and V. Poor, "Abstract dynamic programming models under commutativity conditions," *SIAM J. Control and Optimization*, vol. 25 (July 1987), pp. 990-1006.

37. A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-13 (Apr. 1967), pp. 260–269.

38. Y. Weiss, "Correctness of local probability propagation in graphical models with loops," submitted to *Neural Computation*, July 1998.

39. N. Wiberg, *Codes and Decoding on General Graphs.* Linköping Studies in Science and Technology, Dissertations no. 440. Linköping, Sweden, 1996.

40. M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM J. Alg. Discrete Methods*, vol. 2 (1981), pp. 77–79.