Probability Propagation

by

Glenn R. Shafer and Prakash P. Shenoy

January, 1989

School of Business, University of Kansas, Summerfield Hall
Lawrence, Kansas 66045-2003, USA

Please direct all correspondence regarding this paper to Prakash P. Shenoy at the above address.
This manuscript is a revised version of:
G. R. Shafer and P. P. Shenoy, Probability propagation, *Preliminary Papers of the Second International Workshop on Artificial Intelligence and Statistics* (Fort Lauderdale, Florida, USA, 1989), 33.1-33.10. Also, Working Paper No. 200, School of Business, University of Kansas, Lawrence, KS, August 1988.
The authors will be grateful for suggestions for improvement.

G. R. Shafer and P. P. Shenoy

TABLE OF CONTENTS

Probability Propagation

by

Glenn R. Shafer and Prakash P. Shenoy

School of Business, University of Kansas, Summerfield Hall,
Lawrence, Kansas, 66045-2003, USA

Abstract

In this paper we give a simple account of local computation of marginal probabilities for when the joint probability distribution is given in factored form and the sets of variables involved in the factors form a hypertree. Previous expositions of such local computation have emphasized conditional probability. We believe this emphasis is misplaced. What is essential to local computation is a factorization. It is not essential that this factorization be interpreted in terms of conditional probabilities. The account given here avoids the divisions required by conditional probabilities and generalizes readily to alternative measures of subjective probability, such Dempster-Shafer or Spohnian belief functions.

Key Words: probability propagation, local computation, hypertree, construction sequence, hypertree cover, Markov tree, array, potential, parallel processing.

1. Introduction

In this paper, we study local computation for probability distributions. More precisely, we describe a simple way marginal probabilities can be computed when a joint probability distribution is given in factored form, and the sets of variables involved in the factors form a hypertree.

The phrase "local computation" refers to a computation that involves only a small number of variables. The adjective "local" is used because the variables involved in a given computation are near each other when the relations among the variables are represented graphically.

The purpose of this paper is to simplify and unify previous work. The basic algorithms we describe in sections 7 and 8 do not go beyond the algorithms of Kelly and Barclay [11], Cannings, Thompson and Skolnick [6], Pearl [19], and Lauritzen and Spiegelhalter [14] in what they accomplish, but they do show that the accomplishment is simpler than sometimes thought. All of these earlier authors emphasized conditional probabilities, and all of their algorithms require divisions in order to compute conditional probabilities. But as we show in this paper, the computation of marginal probabilities from factored joint distributions does not require any divisions or any reference to conditional probability. What is essential to local computation is a factorization. It is not essential that this factorization be interpreted, at any stage, in terms of conditional probabilities. Conditional probabilities can be obtained as a by-product of local computation, and they can often strengthen the intuitive interpretation of intermediate computations. But the divisions needed in order to obtain conditional probabilities are unnecessary if only marginal probabilities are desired (Shafer and Shenoy [24]).

Because our approach does not involve conditional probability, it generalizes readily to measures of belief, such as Dempster-Shafer belief functions, for which conditionals do not play a prominent role. In fact, we first learned the approach in the context of Dempster-Shafer belief functions (Shenoy and Shafer [27], Shenoy, Shafer and Mellouli [31], Shafer, Shenoy and Mellouli [25]).

In Shafer and Shenoy [24], we explain how to abstract the approach given here to a set of axioms that apply not only to probability and belief-function propagation but also to constraint propagation (Seidel [22], Dechter and Pearl [7], Shenoy and Shafer [29]), discrete optimization (Bertele and Brioschi [4], Shenoy and Shafer [30]), solving systems of linear equations (Rose [21]), propagation of Spohnian belief functions (Spohn [32], Hunter [10]), retrieval from acyclic databases (Malvestuto [16], Beeri *et al.* [2]), rule propagation in rule-based systems (Shenoy [26]), and implementation of the Kalman filter (Dempster [8], Meinhold and Singpurwalla [17]).

An outline of this paper is as follows. In section 2, we review some graph-theoretic concepts. In section 3, we introduce a notation for probability distributions and for more general functions that we call potentials and arrays. In section 4, we define marginalization for arrays and potentials, and in section 5, we study multiplication and factorization of arrays.

In section 6, we show how local computation can be used to marginalize a factorization on a hypergraph to the smaller hypergraph resulting from the deletion of a twig. Once we know how to delete a twig, we can reduce a hypertree to a single hyperedge by successively deleting twigs. When we have reduced a factorization on a hypertree to a factorization on a single hyperedge, it is no longer a factorization; it is simply the marginal for the hyperedge.

In section 7, we shift our attention from the hypertree to the Markov tree determined by a branching for the hypertree. Using this Markov tree, we describe more graphically the process of marginalizing to a single hyperedge. Our description is based on the idea that each vertex in the tree is a processor, which can operate on arrays for the variables it represents and then send the result to a neighboring processor. In section 8, we generalize this idea to a scheme of simultaneous computation and message passing that produces marginals for all the vertices in the Markov tree. Finally, in section 9, we illustrate our propagation scheme by means of a numerical example.

Our treatment of local computation applies to arrays in general, not just to probability distributions. We take this approach not because the greater generality is of practical importance, but rather because it distances us from probabilistic interpretations and allows us to concentrate on purely computational aspects of our problem. In particular, it frees us from the temptation to seek a probabilistic interpretation for every step in the computation.

In Shafer and Shenoy [24], we explore the connections between factorizations of the joint probability distribution and probabilistic notions of conditional probability and conditional independence. We show that the algorithm of section 8 applied to probability trees results in the generalization of Bayes' theorem developed by Kelly and Barclay [11] and Pearl [19]. Also, we show that Lauritzen and Spiegelhalter's [14] algorithm differs only slightly from the algorithm of section 8.

2. Some Concepts from Graph Theory

Most of the concepts reviewed here have been studied extensively in the graph theory literature (see Berge [3], Golumbic [9], and Maier [15]). A number of the terms we use are new, however - among them, *hypertree*, *construction sequence*, *branch*, *twig, bud,* and *Markov tree*. A *hypertree* is what other authors have called an acyclic (Maier [15]) or decomposable (Lauritzen, Speed, and Vijayan [13]) hypergraph. A *construction sequence* is what other authors have called a sequence with the running intersection property. A *Markov tree* is what authors in database theory have called a join tree (see Maier [15]). We have borrowed the term *Markov tree* from probability theory, where it means a tree of variables in which separation implies probabilistic conditional independence given the separating variables. We first used the term in a non-probabilistic context in Shenoy and Shafer [27] and in Shafer, Shenoy, and Mellouli [25], where we justified it in terms of a concept of qualitative independence analogous to probabilistic independence.

Hypergraphs and Hypertrees. We call a non-empty set $\mathcal{H}$ of non-empty subsets of a finite set $\mathcal{X}$ a *hypergraph* on $\mathcal{X}$. We call the elements of $\mathcal{H}$ *hyperedges*. We call the elements of $\mathcal{X}$ *vertices.*

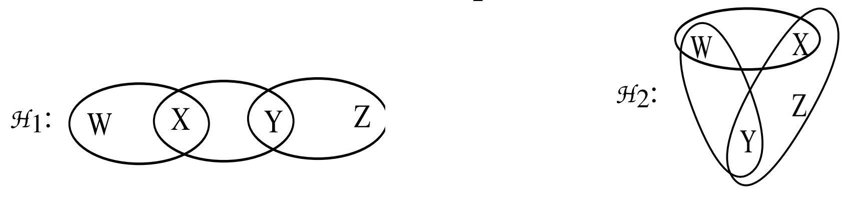
Suppose t and b are distinct hyperedges in a hypergraph $\mathcal{H}$, $t \cap b \neq \varnothing$, and b contains every vertex of t that is contained in a hyperedge of $\mathcal{H}$ other than t; $t \cap (\cup(\mathcal{H}-t)) \subseteq t \cap b$.

Then we call t a *twig* of $\mathcal{H}$, and we call b a *branch* for t. A twig may have more than one branch.

We call a hypergraph a *hypertree* if there is an ordering of all its hyperedges, say $h_1h_2...h_n$, such that h_k is a twig in the hypergraph $\{h_1,h_2,...,h_k\}$ whenever $2 \le k \le n$. We call any such ordering of the hyperedges a *hypertree construction sequence* for the hypertree. We call the first hyperedge in a hypertree construction sequence the *root* of the hypertree construction sequence.

Figure 1 illustrates hypergraphs, hypertrees, twigs and construction sequences.

Figure 1. Two hypergraphs on {W,X,Y,Z}. The hypergraph $\mathcal{H}_1$ is a hypertree, hyperedges {W,X} and {Y,Z} are twigs, and {W,X}{X,Y} {Y,Z} is a construction sequence. The hypergraph $\mathcal{H}_2$ is not a hypertree and it has no twigs.



Since each hyperedge we add as we construct a hypertree is a twig when it is added, it has at least one branch in the hypertree at that point. Suppose we choose such a branch, say $\beta(h)$, for each hyperedge h we add. By doing so, we define a mapping β from $\mathcal{H}$-$\{h_1\}$ to $\mathcal{H}$, where h_1 is the root of the hypertree construction sequence. We will call this function a *branching* for the hypertree construction sequence.

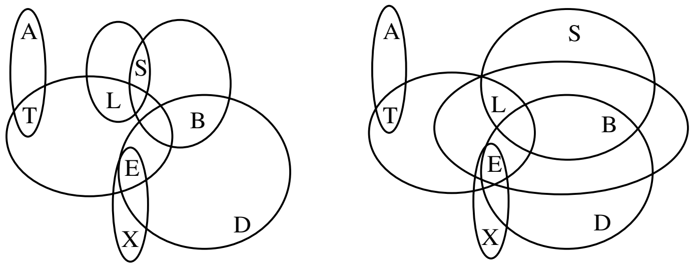
Since a twig may have more than one branch, a hypertree construction sequence may have more than one branching. In general, a hypertree will have many construction sequences. In fact, for each hyperedge of a hypertree, there is a construction sequence beginning with that hyperedge.

Hypertree Covers of Hypergraphs. As we will show, local computation requires two things. The joint probability distribution with which we are working must factor into functions each involving a small set of variables. And these sets of variables must form a hypertree.

If the sets of variables form instead a hypergraph that is not a hypertree, then we must enlarge it until it is a hypertree. We can talk about this enlargement in two different ways. We can say we are adding larger hyperedges, keeping the hyperedges already there. Or, alternatively, we can say we are replacing the hyperedges already there with larger hyperedges. The choice between these two ways of talking does not matter much, because the presence of superfluous twigs (hyperedges contained in other hyperedges) does not affect whether a hypergraph is a hypertree, and because the computational cost of the procedures we will be describing depends primarily on the size of the largest hyperedges, not on the number of the smaller hyperedges (Kong [12]).

We will say that a hypergraph $\mathcal{H}^*$ *covers* a hypergraph $\mathcal{H}$ if for every h in $\mathcal{H}$ there is an element h* of $\mathcal{H}^*$ such that $h \subseteq h^*$. We will say that $\mathcal{H}^*$ is a *hypertree cover* for $\mathcal{H}$ if $\mathcal{H}^*$ is a hypertree and it covers $\mathcal{H}$. Figure 2 shows a hypergraph that is not a hypertree and a hypertree cover for it.

Figure 2. *Left:* A hypergraph that is not a hypertree. *Right*: A hypertree cover for it obtained by replacing hyperedges {S,L} and {S,B} with hyperedges {S,L,B} and {L,E,B}.



Finding a hypertree cover is never difficult. The hypertree $\{\mathcal{X}\}$, which consists of the single hyperedge $\mathcal{X}$, is a hypertree cover for any hypergraph on $\mathcal{X}$. The problem of finding a hypertree cover whose largest hyperedge is as small as possible is NP-complete (Arnborg, Corneil and Proskurowski [1]). Heuristics for finding good hypertree covers is the subject of a growing literature; see e.g., Rose [20], Bertele and Brioschi [4], Tarjan and Yannakakis [33], Kong [12], Mellouli [18], and Zhang [34]. This paper makes no contribution to this problem. Our purpose is rather to explain the process of finding marginals using local computation once a factorization relative to a hypertree is in place.

Trees. A *graph* is a pair $(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is a non-empty set and $\mathcal{E}$ is a set of two-element subsets of $\mathcal{V}$. We call the elements of $\mathcal{V}$ *vertices*, and we call the elements of $\mathcal{E}$ *edges*.

Suppose $(\mathcal{V},\mathcal{E})$ is a graph. If {v,v'} is an element of $\mathcal{E}$, then we say that v and v' are *neighbors*. We call a vertex of a graph a *leaf* if it is contained in only one edge, and we call the other vertex in that edge the *bud* for the leaf. If $v_1v_2...v_n$ is a sequence of distinct vertices, where n>1, and $\{v_k,v_{k+1}\} \in \mathcal{E}$ for k=1,2,...,n-1, then we call $v_1v_2...v_n$ a *path from v_1 to v_n*.

We call a graph a *tree* if there is an ordering of its vertices, say $v_1v_2...v_n$ such that v_k is a leaf in the graph $(\{v_1,v_2,...,v_k\},\mathcal{E}_k)$, where $\mathcal{E}_k$ is the subset of $\mathcal{E}$ consisting of those edges that contain only vertices in $\{v_1,v_2,...,v_k\}$. We call any such ordering of the vertices a *tree construction sequence* for the tree. We call the first vertex in a tree construction sequence the *root* of the tree construction sequence.

Since each vertex we add as we construct a tree is a leaf when it is added, it has a bud in the tree at that point. Given a tree construction sequence and a vertex v that is not the root, let $\beta(v)$ denote the bud for v as it is added. This defines a mapping β from $\mathcal{V}$-$\{v_1\}$ to $\mathcal{V}$, where v_1 is the root. We will call this mapping the *budding* for the tree construction sequence.

The budding for a tree construction sequence is analogous to the branching for a hypertree construction sequence, but there are significant differences. Whereas there may be many branchings for a given hypertree construction sequence, there is only one budding for a given tree construction sequence. In fact, there is only one budding with a given root.

Markov Trees. We call a tree $(\mathcal{H},\mathcal{E})$ a *Markov tree* if the following conditions are satisfied:

(i) $\mathcal{H}$ is a hypergraph.

(ii) If $\{h,h'\}\in\mathcal{E}$, then $h\cap h'\neq\varnothing$.

(iii) If h and h' are distinct vertices, and X is in both h and h', then X is in every vertex on the path from h to h'.

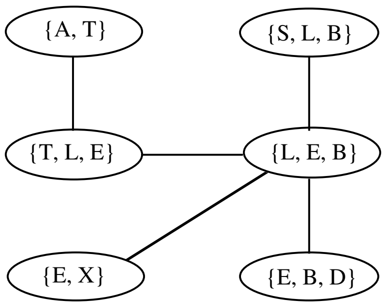
This definition does not state that $\mathcal{H}$ is a hypertree, but it implies that it is:

Proposition 1. (i) If $(\mathcal{H},\mathcal{E})$ is a Markov tree, then $\mathcal{H}$ is a hypertree. Any leaf in $(\mathcal{H},\mathcal{E})$ is a twig in $\mathcal{H}$. If $h_1h_2...h_n$ is a tree construction sequence for $(\mathcal{H},\mathcal{E})$, with β as its budding, then $h_1h_2...h_n$ is also a hypertree construction sequence for $\mathcal{H}$, with β as a branching. (ii) If $\mathcal{H}$ is a hypertree, $h_1h_2...h_n$ is a hypertree construction sequence for $\mathcal{H}$, and β is a branching for $h_1h_2...h_n$, then $(\mathcal{H},\mathcal{E})$ is a Markov tree, where $\mathcal{E} = \{(h_2,\beta(h_2)),...,(h_n,\beta(h_n))\}$; $h_1h_2...h_n$ is a tree construction sequence for $(\mathcal{H},\mathcal{E})$, and β is its budding.

See Shafer and Shenoy [24] for a proof of proposition 1.

If $(\mathcal{H},\mathcal{E})$ is a Markov tree, then we call $(\mathcal{H},\mathcal{E})$ a *Markov tree representative* for the hypertree $\mathcal{H}$. As per proposition 1, every hypertree has a Markov tree representative. Most hypertrees have more than one. Figure 3 shows a Markov tree representative for the hypertree in figure 2.

Figure 3. A Markov tree representative for the hypertree in Figure 2.



Notice that as we delete leaves from a Markov tree (a visually transparent operation), we are deleting twigs from the hypertree.

3. Arrays, Potentials, and Probability Distributions

We use the symbol $\mathcal{W}_X$ for the set of possible values of a variable X, and we call $\mathcal{W}_X$ the *frame* for X. Given a non-empty set h of variables, we let $\mathcal{W}_h$ denote the Cartesian product of $\mathcal{W}_X$ for X in h; we call $\mathcal{W}_h$ the *frame* for h. We will consider only variables with finite frames and only finite sets of variables.

We will call elements of $\mathcal{W}_h$ *configurations of h.* We will use lower-case, bold-faced letters such as **x**, **y**, etc. to denote configurations. If **x** is a configuration of h, **y** is a configuration of g, and $h \cap g = \varnothing$, then $(\mathbf{x},\mathbf{y})$ is a configuration of $h \cup g$.

We call any real-valued function on $\mathcal{W}_h$ an *array* on h. An array is a *potential* if its values are non-negative and not all zero. A potential is a *probability distribution* if its values add to one.

4. Marginalizing Arrays

Marginalization, familiar from probability theory, means reducing a function on one set of variables to a function on a smaller set of variables by summing over the variables omitted.

Suppose g and h are sets of variables, $h \subseteq g$, and G is an array on g. The *marginal* of G on h, denoted by $G^{\downarrow h}$, is an array on h. It is defined by

$$G^{\downarrow h}(\mathbf{x}) = \Sigma\{G(\mathbf{x},\mathbf{y}) \mid \mathbf{y} \in \mathcal{W}_{g-h}\}$$

for all $\mathbf{x} \in \mathcal{W}_h$. For example, if G is an array on the variables {W,X,Y,Z}, then the marginal $G^{\downarrow\{W,X\}}$ is given by $G^{\downarrow\{W,X\}}(w,x) = \Sigma\{G(w,x,y,z) \mid (y,z) \in \mathcal{W}_{\{Y,Z\}}\}$.

If $k \subseteq h \subseteq g$ and G is an array on g, then $(G^{\downarrow h})^{\downarrow k} = G^{\downarrow k}$.

When $h \subseteq g$ and P is a probability distribution on g, the marginal $P^{\downarrow h}$ is P's marginal on h in the usual probabilistic sense; $P^{\downarrow h}(\mathbf{x})$ is the probability that the variables in h take the values in **x**.

5. Multiplying and Factoring Arrays

In order to develop a notation for the multiplication of arrays, we first need a notation for the projection of configurations. Here projection means dropping extra coordinates; if (w,x,y,z) is a configuration of {W,X,Y,Z}, for example, then the projection of (w,x,y,z) to {W,X} is simply (w,x), which is a configuration of {W,X}. If g and h are non-empty sets of variables, $h \subseteq g$, and $\mathbf{x} \in \mathcal{W}_g$, then we will let $\mathbf{x}^{\downarrow h}$ denote the projection of **x** to h. Note that $\mathbf{x}^{\downarrow h} \in \mathcal{W}_h$.

Multiplication. When we refer to multiplication of arrays, we mean pointwise multiplication; if G and H are arrays on g and h respectively, then their product GH is the array on $g \cup h$ given by $(GH)(\mathbf{x}) = G(\mathbf{x}^{\downarrow g})H(\mathbf{x}^{\downarrow h})$ for all $\mathbf{x} \in \mathcal{W}_{g \cup h}$. If G and H are potentials, their product GH need not be a potential; it is possible that $(GH)(\mathbf{x}) = 0$ for all $\mathbf{x} \in \mathcal{W}_{g \cup h}$.

Factorization. Suppose A is an array on a finite set of variables $\mathcal{X}$, and suppose $\mathcal{H}$ is a hypergraph on $\mathcal{X}$. If A is equal to the product of arrays on the hyperedges of $\mathcal{H}$, say $A = \Pi\{A_h \mid h \in \mathcal{H}\}$, where A_h is an array on h, then we say that A *factors on* $\mathcal{H}$.

When A does factor on $\mathcal{H}$, the arrays A_h are not unique. We can multiply one of the A_h by a non-zero constant if we compensate by dividing another by the same constant. More generally, if g and h overlap, then we can multiply A_g and divide A_h by any array on $g \cap h$ that has no zero values.

When an array factors on a hypergraph, it also factors on any larger hypergraph. More generally, when an array A on $\mathcal{X}$ factors on a hypergraph $\mathcal{H}$ on $\mathcal{X}$, it also factors on any hypergraph $\mathcal{H}^*$ on $\mathcal{X}$ that covers $\mathcal{H}$.

Though the theory in this chapter applies to arrays in general, we will be interested in practice in factorizations of probability distributions. Then means that we will be concerned primarily with arrays that are potentials, for when a probability distribution P factors on a hypergraph $\mathcal{H}$, the arrays A_h in the factorization can be assumed to be potentials. Indeed, since P is not identically zero, none of the A_h can be identically zero. And we can assume that none of the values of the A_h are negative. Since P does not take any negative values, we could change the sign of any negatives values of the A_h without changing the validity of the factorization.

After we condition on observations, we are often left working with factorizations of potentials that are proportional to probability distributions that interest us. Suppose, indeed, that we begin with a factorization of a joint probability distribution on $\mathcal{H}$; $P = \Pi\{A_h \mid h\in\mathcal{H}\}$. If we observe the values of the variables in g, say $\mathbf{y}\in\mathcal{W}_g$, then we will be interested in the conditional distribution $P^{|g=\mathbf{y}}$, which will be proportional to $(\Pi\{A_h \mid h\in\mathcal{H}\})\, I^{g=\mathbf{y}}$, where $I^{g=\mathbf{y}}$ is the potential on g given by $I^{g=\mathbf{y}}(\mathbf{x}) = 0$ if $\mathbf{x}\neq\mathbf{y}$, and $I^{g=\mathbf{y}}(\mathbf{x}) = 1$ if $\mathbf{x}=\mathbf{y}$. We call $I^{g=\mathbf{y}}$ the *indicator potential for* $g=\mathbf{y}$.

The following proposition plays a key role in making local computation possible for propagation of probabilities.

Proposition 2. Suppose G and H are arrays on g and h respectively, and $g\cap h\neq\emptyset$. Then $(GH)^{\downarrow g} = G(H^{\downarrow g\cap h})$.

The result stated in proposition 2 follows directly from the definitions of multiplication and marginalization of arrays.

6. Marginalizing Factorizations

In this section, we learn how to adjust a factorization on a hypergraph to account for the deletion of a twig. This can be accomplished by local computation, computation involving only the arrays on the twig and a branch for the twig.

Suppose $\mathcal{H}$ is a hypergraph on $\mathcal{X}$, t is a twig in $\mathcal{H}$, and b is a branch for t. The twig t may contain some vertices that are not contained in any other hyperedge in $\mathcal{H}$. These are the vertices in the set t-b. Deleting t from $\mathcal{H}$ means reducing $\mathcal{H}$ to the hypergraph $\mathcal{H}$-{t} on the set $\mathcal{X}'=\mathcal{X}-(t-b)$.

Suppose A is an array on $\mathcal{X}$, suppose A factors on $\mathcal{H}$, and suppose we have stored A in factored form, i.e., $A = \Pi\{A_h \mid h\in\mathcal{H}\}$. The following proposition tells us how to adapt this factorization to a factorization of $A^{\downarrow\mathcal{X}'}$ on $\mathcal{H}$-{t}, with a computation that involves only t and its branch.

Proposition 3. Under the assumptions of the preceding paragraph,

$$A^{\downarrow\mathcal{X}'} = (\Pi\{A_h \mid h\in\mathcal{H}-\{t,b\}\})(A_b A_t^{\downarrow t\cap b}), \tag{6.1}$$

where b is any branch for t. Thus the marginal $A^{\downarrow\mathcal{X}'}$ factors on the hypergraph $\mathcal{H}$-{t}. The potential on b is multiplied by $A_t^{\downarrow t\cap b}$, and the potentials on the other elements of $\mathcal{H}$-{t} are unchanged.

The result stated in proposition 3 follows directly from proposition 2 by letting $\Pi\{A_h \mid h\in\mathcal{H}-\{t\}\} = G$ and $A_t = H$.

Proposition 3 is especially interesting in the case of hypertrees, because repeated application of (6.1) allows us to obtain A's marginal on any particular hyperedge of $\mathcal{H}$. If we want the marginal on a hyperedge h_1, we choose a construction sequence beginning with h_1, say $h_1h_2...h_n$. Let $\mathcal{X}_k$ denote $h_1\cup...\cup h_k$, and let $\mathcal{H}_k$ denote $\{h_1, h_2, ..., h_k\}$, for k=1,...,n-1. We use (6.1) to delete the twig h_n, so that we have a factorization of $A^{\downarrow\mathcal{X}_{n-1}}$ on the hypertree $\mathcal{H}_{n-1}$. Then we use (6.1) again to delete the twig h_{n-1} so that we have a factorization of $A^{\downarrow\mathcal{X}_{n-2}}$ on the hypertree $\mathcal{H}_{n-2}$. And so on, until we have deleted all the hyperedges except h_1, so that we have a factorization of $A^{\downarrow h_1}$ on the hypertree $\{h_1\}$, i.e., we have the marginal $A^{\downarrow h_1}$. At each step, the computation is local, in the sense that it involves only a twig and its branch.

We are most interested, of course, in the case where A is a probability distribution. In this case, as we mentioned in the preceding section, the factorization we wish to marginalize may be a proportionality rather than an equality. In other words, we may begin with a factorization of a potential that is only proportional to the probability distribution that interest us. Eventually, we will need to find the constant of proportionality, but since marginalization preserves proportionality, we may postpone the normalization until the final step, where we have reduced the potential to its marginal on the single hyperedge with which we are concerned, and hence normalization requires summation only over the frame for this hyperedge.

7. Computing Marginals in Markov Trees

As we learned in section 2, the choice of a branching for a hypertree determines a Markov tree for the hypertree. We now look at our scheme for computing a marginal from the viewpoint of this Markov tree. This change in viewpoint does not necessarily affect the implementation of the computation, but it gives us a richer understanding. It gives us a picture in which message passing, instead of deletion, is the dominant metaphor, and in which we have great flexibility in how the message passing is controlled.

Why did we talk about deleting the hyperedge h_k as we projected h_k's array to the branch $\beta(h_k)$? The point was simply to remove h_k from our attention. The "deletion" had no computational significance, but it helped make clear that h_k and the array on it were of no further use. What was of further use was the smaller hypertree that would remain were h_k deleted.

When we turn from the hypertree to the Markov tree, deletion of twigs translates into deletion of leaves. But a tree is easier to visualize than a hypertree. We can remove a leaf or a whole branch of a tree from our attention without leaning so heavily on metaphorical deletion. And a Markov tree also allows another, more useful, metaphor. We can imagine that each vertex of the tree is a processor, and we can imagine that the projection is a message that one processor passes to another. Within this metaphor, vertices no longer relevant are kept out of our way by the rules guiding the message passing, not by deletion.

Translating to the Markov Tree. The algorithm of the preceding section requires a hypertree construction sequence $h_1h_2...h_n$ and a branching β for $h_1h_2...h_n$. We repeatedly apply

> *Operation H.* Marginalize the array now on h_k to $\beta(h_k)$. Change the array now on $\beta(h_k)$ by multiplying it by this marginalization.

We apply Operation H first for k=n, then for k=n-1, and so on, down to k=2. The array assigned to h_1 at the end of this process is the marginal on h_1.

Now consider the Markov tree $(\mathcal{H},\mathcal{E})$ determined by the branching β. The vertices of $(\mathcal{H},\mathcal{E})$ are the hyperedges h_1, h_2, ..., h_n. We imagine that a processor is attached to each h_i. The processor attached to h_i can store an array defined on h_i, can compute the marginalization of this array to h_j, where h_j is a neighboring vertex, can send the marginalization to h_j as a message, can accept an array on h_i as a message from a neighbor, and can change the array it has stored by multiplying it by such an incoming message.

The edges of $(\mathcal{H},\mathcal{E})$ are $\{h_n,\beta(h_n)\}$, $\{h_{n-1},\beta(h_{n-1})\}$, ..., $\{h_3,\beta(h_3)\}$, $\{h_2,h_1\}$. When we move from h_n to $\beta(h_n)$, then from h_{n-1} to $\beta(h_{n-1})$, and so on, we are moving inwards in the Markov tree, from the outer leaves to the root h_1. The repeated application of Operation H by the processors located at the vertices follows this path.

Let Cur_h denote the array currently stored by the processor at vertex h of $(\mathcal{H},\mathcal{E})$. In terms of the local processors and the Cur_h, Operation H becomes the following:

> *Operation M_1.* Vertex h computes $\mathrm{Cur}_h^{\downarrow h\cap\beta(h)}$, the marginalization of Cur_h to $h\cap\beta(h)$. It sends $\mathrm{Cur}_h^{\downarrow h\cap\beta(h)}$ as a message to vertex $\beta(h)$. Vertex $\beta(h)$ accepts the message $\mathrm{Cur}_h^{\downarrow h\cap\beta(h)}$ and changes $\mathrm{Cur}_{\beta(h)}$ by multiplying it by $\mathrm{Cur}_h^{\downarrow h\cap\beta(h)}$.

At the outset, $\mathrm{Cur}_h = A_h$ for every vertex h. Operation M_1 is executed first for $h=h_n$, then for $h=h_{n-1}$, and so on, down to $h=h_2$. At the end of this propagation process, the array Cur_{h_1}, the array stored at h_1, is the marginal of A on h_1.

An Alternative Operation. Operation M_1 prescribes actions by two processors, h and $\beta(h)$. We now give an alternative, Operation M_2, which is executed by a single processor. Operation M_2 differs from Operation M_1 only in that it requires a processor to multiply together the messages it receives all at once, rather than incorporating them into the product one by one as they arrive.

Operation M_{2a}. Vertex h multiplies the array A_h by all the messages it has received, and it calls the result Cur_h. Then it computes $Cur_h^{\downarrow h\cap\beta(h)}$, the marginalization of Cur_h to $h\cap\beta(h)$. It sends $Cur_h^{\downarrow h\cap\beta(h)}$ as a message to $\beta(h)$.

Operation M_{2a} involves action by only one processor, the processor h. When Operation M_{2a} is executed by h_n, there is no multiplication, because h_n, being a leaf in the Markov tree, has received no messages. The same is true for the other leaves in the Markov tree. But for vertices that are not leaves in the Markov tree, the operation will involve both multiplication and marginalization.

After Operation M_{2a} has been executed by h_n, h_{n-1}, and so on down to h_2, the root h_1 will have received a number of messages but will not yet have acted. To complete the process, h_1 must multiply together all its messages and its original array A_{h_1}, thus obtaining the marginal $A^{\downarrow h_1}$. We may call this Operation M_{2b}:

Operation M_{2b}. Vertex h_1 multiplies the array A_{h_1} by all the messages it has received, and it reports the result to the user of the system.

Operation M_2 simplifies our thinking about control, or the flow of computation, because it allows us to think of control as moving with the computation in the Markov tree. In our marginalization scheme, control moves from one vertex to another, from the outer leaves inward towards the root. If we use Operation M_2, then a vertex is computing only when it has control.

Formulas for the Messages. We have described verbally how each vertex computes the message it sends to its branch. Now we will translate this verbal description into a formula that constitutes a recursive definition of the messages.

Let $M^{h\to\beta(h)}$ denote the message sent by vertex h to its bud. Our description of Operation M_{2a} tells us that $M^{h\to\beta(h)} = Cur_h^{\downarrow h\cap\beta(h)}$, where $Cur_h = A_h \prod\{M^{g\to\beta(g)} \mid g\in\mathcal{H}$ and $\beta(g)=h\}$. Putting these two formulas together, we have

$$M^{h\to\beta(h)} = (A_h \prod\{M^{g\to\beta(g)} \mid g\in\mathcal{H} \text{ and } \beta(g)=h\})^{\downarrow h\cap\beta(h)}. \tag{7.1}$$

If h is a leaf, then there is no $g\in\mathcal{H}$ such that $h=\beta(g)$, and so (7.1) reduces to

$$M^{h\to\beta(h)} = A_h^{\downarrow h\cap\beta(h)}, \tag{7.2}$$

by the convention that an empty product is equal to one.

Formula (7.1) constitutes a recursive definition of $M^{h\to\beta(h)}$ for all h, excepting only the root h_1 of the budding β. The special case (7.2) defines $M^{h\to\beta(h)}$ for the leaves; a further application of (7.1) defines $M^{h\to\beta(h)}$ for vertices one step in towards the root from the leaves; a third application defines $M^{h\to\beta(h)}$ for vertices two steps in towards the root from the leaves; and so on.

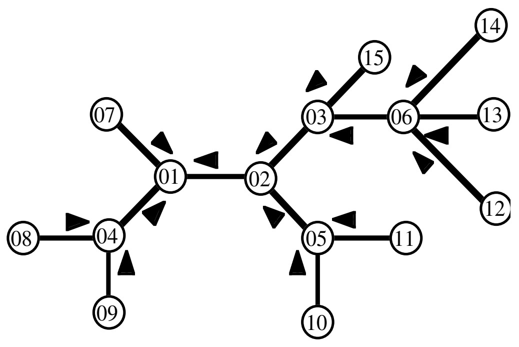
We can also represent Operation M_{2b} by a formula:

$$A^{\downarrow h_1} = A_{h_1}\,\Pi\{M^{g\to\beta(g)} \mid g\in\mathcal{H} \text{ and } \beta(g)=h_1\}. \tag{7.3}$$

Flexibility of Control. Whether we use operation M_1 or M_2, it is not necessary to follow exactly the order h_n, h_{n-1}, and so on. The final result will be the same provided only that a processor never send a message until after it has received and absorbed all the messages it is supposed to receive.

This point is obvious when we look at a picture of the Markov tree. Consider, for example a Markov tree with 15 vertices, as in figure 4. The vertices are numbered from 1 to 15 in this picture, indicating a construction sequence $h_1h_2...h_{15}$. Since we want to find the marginal for vertex 1, all our messages will be sent towards vertex 1, in the directions indicated by the arrows. Our scheme calls for a message from vertex 15 to vertex 3, then a message from vertex 14 to vertex 6, and so on. But we could just as well begin with messages from 10 and 11 to 5, follow with a message from 5 to 2, then messages from 12, 13, and 14 to 6, from 6 and 15 to 3, and so on.

Figure 4. A tree with 15 vertices.

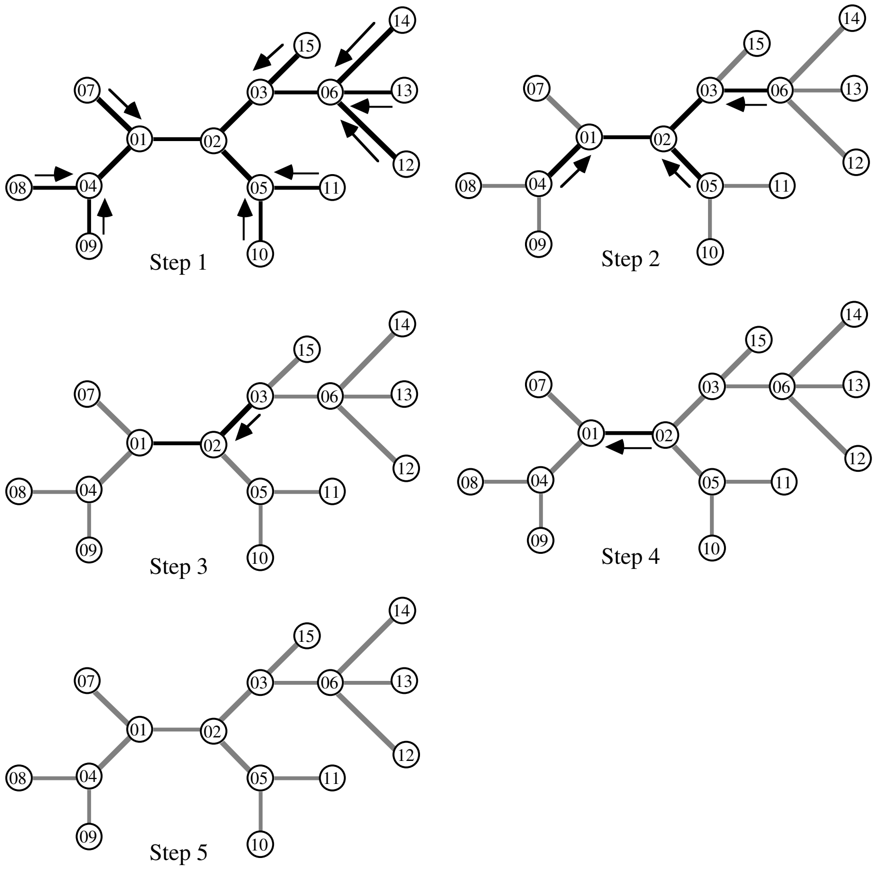


Returning to the metaphor of deletion, where each vertex is deleted when it sends its message, we can say that the only constraint on the order in which the vertices act is that each vertex must be a leaf when it acts; all the vertices that used it as a branch must have sent their messages to it and then been deleted, leaving it a leaf.

The different orders of projection that obey this constraint correspond, of course, to the different tree construction sequences for $(\mathcal{H},\mathcal{E})$ that use the branching β.

So far, we have been thinking about different sequences in which the vertices might act. This is most appropriate if we are really implementing the scheme on a serial computer. But if the different vertices really did have independent processors that could operate in parallel, then some of the vertices could act simultaneously. Figure 5 illustrates one way this might go for the Markov tree of figure 4. In step 1, all the leaf processors project to their branches. In step 2, vertices 4, 5, and 6 (which would be leaves were the original leaves deleted) project. And so on.

Figure 5. An example of the message-passing scheme for computation of the marginal of vertex 1.



If the different processors take different amounts of time to perform Operation M_2 on their inputs, then the lock-step timing of figure 5 may not provide the quickest way to find the marginal for h_1. It may be quicker to allow a processor to act as soon as it receives messages from its leaves, whether or not all the other processors that started along with these leaves have finished.

In general, the only constraint, in the parallel as in the serial case, is that action move inwards towards the root or goal, vertex h_1. Each vertex must receive and absorb all its messages from vertices farther away from h_1 before sending its own message on towards h_1.

If we tell each processor who its neighbors are and which one of these neighbors lies on the path towards the goal, then no further global control or synchronization is needed. Each processor knows that it should send its outgoing message as soon as it can after receiving all its incoming messages. The leaf processors, which have no incoming messages, can act immediately. The others must wait their turn.

A Simple Production System. In reality, we will never have a parallel computer organized precisely to fit our problem. Our story about passing messages between independent processors should be thought of as metaphor, not as a guide to implementation. Implementations can take advantage, however, of the modularity the metaphor reveals.

One way to take advantage of this modularity, even on a serial computer, is to implement the computational scheme in a simple forward-chaining production system. A forward-chaining production system consists of a working memory and a rule-base, a set of rules for changing the contents of the memory. (See Brownston *et al.* [5]).

A very simple production system is adequate for our problem. We need a working memory that initially contains A_h for each vertex h of $(\mathcal{H},\mathcal{E})$, and a rule-base consisting of just two rules, corresponding to Operations M_{2a} and M_{2b}.

Rule 1: If A_h is in working memory and $M^{g\rightarrow\beta(g)}$ is in working memory for every g such that $\beta(g)=h$, then use (7.1) to compute $M^{h\rightarrow\beta(h)}$, and place it in working memory.

Rule 2: If A_{h_1} is in working memory and $M^{g\rightarrow\beta(g)}$ is in working memory for every g such that $\beta(g)=h_1$, then use (7.3) to compute $A^{\downarrow h_1}$, and print the result.

Initially, there will be no $M^{g\rightarrow\beta(g)}$ at all in working memory, so Rule 1 can fire only for h such that there is no g with $\beta(g)=h$ - i.e., only for h that are leaves. But eventually Rule 1 will fire for every vertex except the root h_1. Then Rule 2 will fire, completing the computation. Altogether, there will be n firings, one for each vertex in the Markov tree.

Production systems are usually implemented so that a rule will fire only once for a given instantiation of its antecedent; this is called *refraction* (Brownston *et al.* [5, pp. 62-63]). If our simple production system is implemented with refraction, there will be no unnecessary firings of rules; only the n firings that are needed will occur. Even without refraction, however, the computation will eventually be completed.

Since refraction allows a rule to fire again for a given instantiation when the inputs for that instantiation are changed, this simple production system will also handle updating efficiently, performing only those recomputations that are necessary.

8. Simultaneous Propagation in Markov Trees

In the preceding section, we were concerned with the computation of the marginal on a single vertex of the Markov tree. In this section, we will be concerned with how to compute the marginals on all vertices simultaneously.

Computing all the Marginals. If we can compute the marginal of A on one hyperedge in $\mathcal{H}$, then we can compute the marginals on all the hyperedges in $\mathcal{H}$. We simply compute them one after the other. It is obvious, however, that this will involve much duplication of effort. How can we avoid the duplication?

Notice first that we only need one Markov tree. Though there may be many Markov tree representatives for $\mathcal{H}$, any one of them can serve for the computation of all the marginals. Once we have chosen a Markov tree representative $(\mathcal{H},\mathcal{E})$, then no matter which element h of $\mathcal{H}$ interests us, we can choose a tree construction sequence for $(\mathcal{H},\mathcal{E})$ that begins with h, and since this sequence is also a hypertree construction sequence for $\mathcal{H}$, we can apply the method of section 7 to it to compute $A^{\downarrow h}$.

Notice also that the message passed from one vertex to another, say from f to g, will be the same no matter what marginal we are computing. If β is the budding that we use to compute $A^{\downarrow h}$, the marginal on h, and β' is the budding we use to compute $A^{\downarrow h'}$, and if $\beta(f)=\beta'(f)=g$, then the message $M^{f\rightarrow\beta(f)}$ that we send from f to g when computing $A^{\downarrow h}$ is the same as the message $M^{f\rightarrow\beta'(f)}$ that we send from f to g when computing $A^{\downarrow h'}$. So we may write $M^{f\rightarrow g}$ instead of $M^{f\rightarrow\beta(f)}$ when $\beta(f)=g$.

If we compute marginals for all the vertices, then we will eventually compute both $M^{f\rightarrow g}$ and $M^{g\rightarrow f}$ for every edge {f,g}.

We can easily generalize the recursive definition of $M^{g\rightarrow\beta(g)}$ that we gave in section 7 to a recursive definition of $M^{g\rightarrow h}$ for all neighbors g and h. To do so, we merely restate

(7.1) in a way that replaces references to the budding β by references to neighbors and the direction of the message. We obtain

$$M^{g\rightarrow h} = (\ A_g \prod\{M^{f\rightarrow g} \mid f\in(\mathcal{N}_g\text{-}\{h\})\}\ )^{\downarrow h}, \tag{8.1}$$

where $\mathcal{N}_g$ is the set of all g's neighbors in $(\mathcal{H},\mathcal{E})$. If g is a leaf vertex, then (8.1) reduces to $M^{g\rightarrow h} = A_g{}^{\downarrow h}$.

After we carry out the recursion to compute $M^{g\rightarrow h}$ for all pairs of neighbors g and h, we can compute the marginal of A on each h by

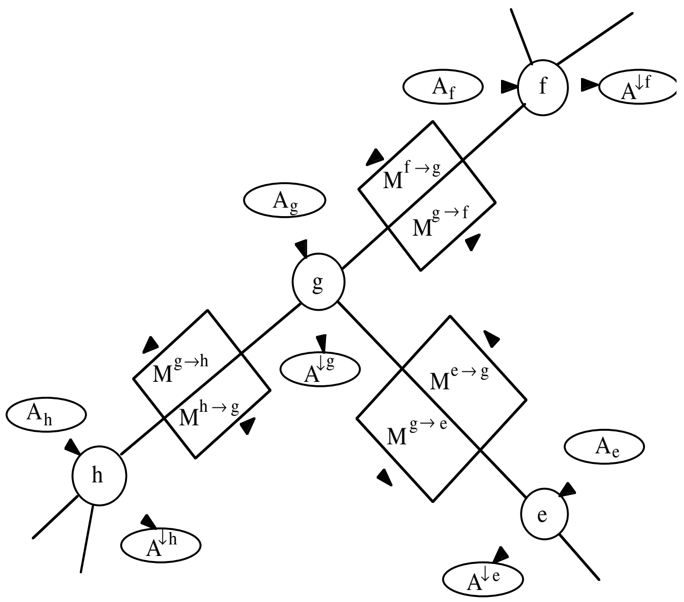
$$A^{\downarrow h} = A_h \prod\{M^{g\rightarrow h} \mid g\in\mathcal{N}_h\}. \tag{8.2}$$

There is exactly twice as much message passing in our scheme for simultaneous computation as there was in our scheme for computing a single marginal. Here every pair of neighbors exchange messages; there only one message was sent between every pair of neighbors. Notice also that we can make the computation of any given marginal the beginning of the simultaneous computation. We can single out any hyperedge h (even a leaf), and forbid it to send a message to any neighbor until it has received messages from all its neighbors. At that point, h can compute its marginal and can also send messages to all its neighbors; the second half of the message passing then proceeds, with messages moving back in the other direction.

The General Architecture. To implement (8.1) and (8.2), we must imagine that our processors have a way to store incoming messages. We simply have two storage registers between every pair of neighbors g and h. One register stores the message from g to h; the other stores the message from h to g.

Figure 6 shows an architecture for the simultaneous computation. In addition to the storage registers that communicate between vertices, this figure shows registers where the original arrays, the A_h, are put into the system and the marginals, the $A^{\downarrow h}$, are read out.

Figure 6. Several vertices, with storage registers for communication between themselves and with the user.

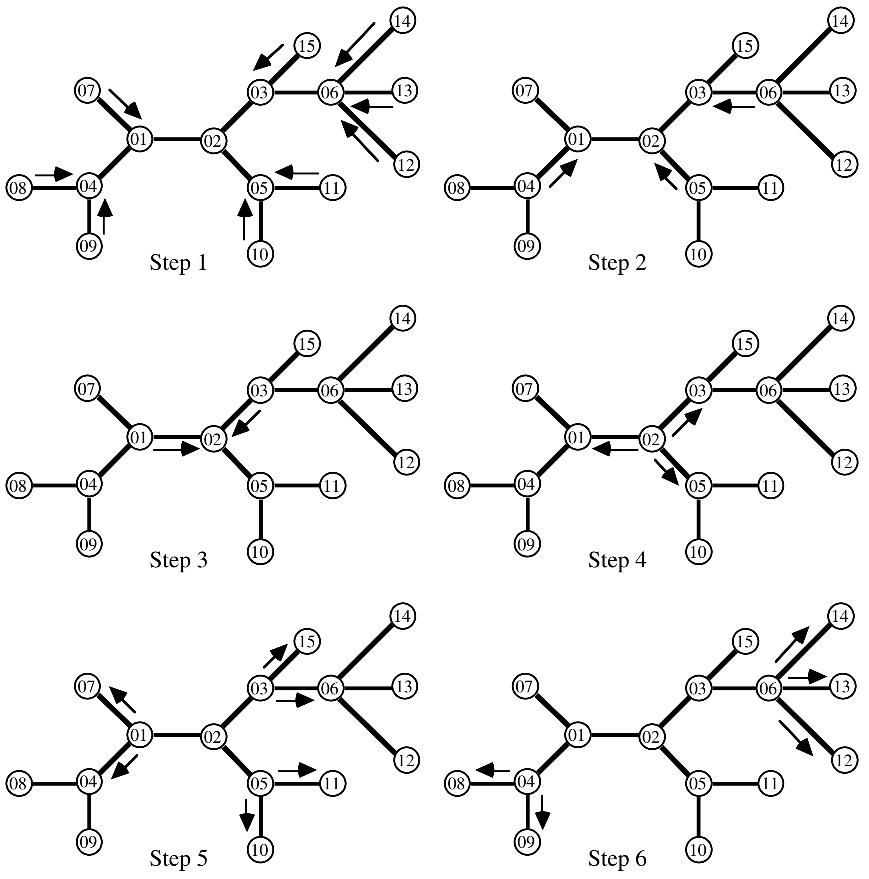


Flexibility of Control. In the architecture of figure 6, computation is controlled by the requirement that a vertex g must have messages in all its incoming registers except the one from h before it can compute a message to send to h.

This basic requirement leaves room for a variety of control regimes. Most of the comments we made about the flexibility of control for the computation of the marginal for a single vertex carry over to figure 6.

It may be enlightening to look at how the lock-step control we illustrated with figure 5 might generalize to simultaneous computation of the marginals for all vertices. Consider a lock-step regime where at each step, each vertex looks and sees what messages it has the information to compute, computes these messages, and sends them. After all the vertices working are done, they look again, see what other messages they now have the information to compute, compute these messages, and send them. And so on. Figure 7 gives an example. At the first step, the only messages that can be computed are the messages from the leaves to their branches. At the second step, the computation moves inward. Finally, at step 3, it reaches vertex 2, which then has the information needed to compute its own marginal and messages for all its neighbors. Then the messages move back out towards the leaves, with each vertex along the way being able to compute its own marginal and messages for all its other neighbors as soon as it receives the message from its neighbor nearest vertex 2.

Figure 7. An example of the message-passing scheme for simultaneous computation of all marginals.



In the first phase, the inward phase, a vertex sends a message to only one of its neighbors, the neighbor towards the center. In the second phase, the outward phase, a vertex sends k-1 messages, where k is the number of its neighbors. Yet the number of messages sent in the two phases is roughly the same, because the leaf vertices participate in the first phase and not in the second.

There are seven vertices in the longest path in the tree of figure 7. Whenever the number of vertices in the longest path is odd, the lock-step control regime will result in computation proceeding inwards to a central vertex and then proceeding back outwards to the leaves. Whenever this number is even, there will instead be two central vertices that send each other messages simultaneously, after which they both send messages back outwards towards the leaves.

If we really do have independent processors for each vertex and we want to get the job done as quickly as possible, we will demand that each processor go to work as quickly as possible subject to this constraint. But the job will get done eventually provided only that all the processors act eventually. It will get done, for example, if each processor checks on its inputs periodically or at random times and acts if it has those inputs (Pearl [19]).

Updating Messages. Suppose we have computed $A^{\downarrow h}$ for each hyperedge h. And suppose we now find reason to change one or more of our inputs, the A_h. If we have implemented the architecture just described, with storage registers between each of the vertices, then we may be able to update the marginals $A^{\downarrow h}$ without discarding all the work we have already done.

Unnecessary computation can be avoided without global control. We simply need a way of marking arrays, to indicate that they have received any needed updating. Suppose the processor at each vertex h can recognize the mark on any of its inputs (on A_h, our direct input, or on any message $M^{g \to h}$ from a neighboring vertex g), and can write the mark on its own output, the message $M^{h \to g}$. When we wish to update the computation of $A^{\downarrow h}$, we put in the new values for those A_h we wish to change, and we mark all the A_h, both the ones we have changed, and the others, which we do not want to change. Then we run the system as before, except that a processor, instead of waiting for its incoming registers to be full before it acts, waits until all its inputs are marked. The processor can recognize when an input is marked without being changed, and in this case it simply marks its output instead of recomputing it.

The idea of updating is important because of conditioning. We often want to condition a probability distribution on the observed values of one or more variables. Conditioning on a variable X can be achieved by multiplying a factorization of the probability distribution by an indicator potential on X. Since this new potential on X can be incorporated in the potential on any hyperedge containing X, conditioning on X can be achieved by changing the input potential in just one of the hyperedges in the hypertree.

If we change just one of the inputs, then efficient updating will save about half the work involved in simply reperforming the entire computation. To see that this is so, consider the effect of changing the input A_h in figure 6. This will change the message $M^{g \to f}$, but not the message $M^{f \to g}$. The same will be true for every edge; one of the two messages will have to be recomputed, but not the other.

The Corresponding Production System. Implementing simultaneous computation in a production system requires only slight changes in our two rules. The following will work:

Rule 1': If A_g is in working memory, $h \in \mathcal{N}_g$, and $M^{f \to g}$ is in working memory for every f in $\mathcal{N}_g$-{h}, then use (8.1) to compute $M^{g \to h}$, and place it in working memory.

Rule 2': If A_h is in working memory, and $M^{g \to h}$ is in working memory for every g in $\mathcal{N}_h$, then use (8.2) to compute $A^{\downarrow h}$, and print the result.

Initially, there will be no $M^{f \to g}$ at all in working memory, so Rule 1' can fire only for g and h such that $\mathcal{N}_g$-{h} is empty - i.e., only when g is a leaf and h is its bud. But eventually Rule 1' will fire in both directions for every edge {g,h}. Once Rule 1' has fired for all the neighbors g of h, in the direction of h, Rule 2' will fire for h. Altogether, there will be 3n-2 firings, two firings of Rule 1' for each of the n-1 edges, and one firing of Rule 2' for each of the n vertices.

As the count of firings indicates, our scheme for simultaneous computation finds marginals for all the vertices with roughly the same effort that would be required to find marginals for three vertices if this were done by running the scheme of section 7 three times.

Relation to Other Work. As we mentioned in the introduction, the algorithm described in this section can be related to the generalization of Bayes' theorem developed by Kelly and Barclay [11] and Pearl [19] and to Lauritzen and Spiegelhalter's [14] algorithm for marginalization of a factored distribution. We will now sketch the relation. For a detailed account, see Shafer and Shenoy [24].

The simplest factored joint probability distributions arise when conditional independence relations allow us to express the distribution as the product of the marginal for one variable, the conditional for a second variable given the first, the conditional for the third given just one of the first two, and so on. In this case, only pairs of variables are involved in the factorization, and they immediately form a hypertree. The most convenient Markov tree is one that includes vertices for both the pairs and the single variables. Begin with a vertex for the first variable alone, say $\{X_0\}$, then for each successive variable, say X_i, attach $\{X_i\}$ to $\{X_i,X_j\}$, where X_j is the variable on which X_i is conditioned, and attach $\{X_i,X_j\}$ to $\{X_j\}$. If we enter the marginal for X_0 on $\{X_0\}$, and the conditional for X_i given $\{X_j\}$. on $\{X_i,X_j\}$, and then propagate, then the simultaneous propagation described in this section will really be propagation downward from X_0 ; the messages sent upwards will be vectors of ones and hence will have no effect. However, if we then enter vectors to indicate observations (for a variable that is observed, we enter a vector that has a one for the observed value and a zero for other values), then the propagation both ways will be meaningful. The messages sent upwards will be likelihoods and the messages sent downwards will be probabilities, just as in Pearl's description.

Lauritzen and Spiegelhalter's general scheme starts with a arbitrary factorization on a hypertree, just as ours does. The propagation is more controlled, however, and messages sent are used immediately and not stored. First we propagate inward to a particular vertex; then we propagate outward from that vertex. On the inward sweep, every message sent is divided out of the array stored at the sender and multiplied into the array stored at the recipient. The division means that no harm will be done when the same information is, in effect sent back (this solves the problem that our algorithm solves by multiplying only messages from other neighbors when computing the message to send to one neighbor). It also has the effect of making the messages stored at the different nodes conditional probabilities; the factorization has in effect been transformed into a factorization into a marginal and conditionals, analogous to the factorization in Pearl's simpler trees. The outward sweep is then analogous to Pearl's downward propagation.

9. An Example

We will now illustrate our propagation scheme using a simple example. The example is adapted from Shachter and Heckerman [23]. Consider three variables D, B and G representing diabetes, blue toe and glucose in urine, respectively. The frame for each variable has two configurations. D=d will represent the proposition *diabetes is present* (in some patient) and D=~d will represent the proposition *diabetes is not present*. Similarly for B and G. Let P denote the joint probability distribution for {D, B, G}. We will assume that diabetes causes blue toe and glucose in urine implying that variables B and G are conditionally independent (with respect to P) given D. Thus we can factor P as follows:

$$P = P^{D}\, P^{B|D}\, P^{G|D} \qquad (9.1)$$

where P^D is the potential on {D} representing the marginal of P for D, $P^{B|D}$ is the potential for {D,B} representing the conditional distribution of B given D, and $P^{G|D}$ is the potential for {D,G} representing the conditional distribution of G given D. For example, $P^{B|D}(d,b)$ represents the conditional probability of the proposition B=b given that D=d. Thus P factors on the hypertree {{D}, {D,B}, {D,G}}. Since we would like to compute the marginals for B and G, we will enlarge the hypertree to include the hyperedges {B} and {G}. It is easy to easy to expand (9.1) so that we have a factorization of P on the enlarged hypertree - the potentials on these additional hyperedges consist of all ones. Suppose that the potentials P^D, $P^{B|D}$, and $P^{G|D}$ are as shown in Table 1.

Table 1. The potentials P^D, $P^{B|D}$, and $P^{G|D}$.

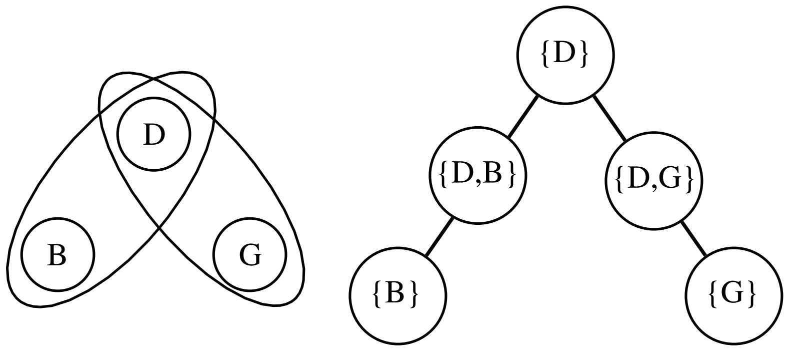
P^D	
d	.1
~d	.9

$P^{B|D}$	
d,b	.014
d,~b	.986
~d,b	.006
~d,~b	.994

$P^{G|D}$	
d,g	.9
d,~g	.1
~d,g	.01
~d,~g	.99

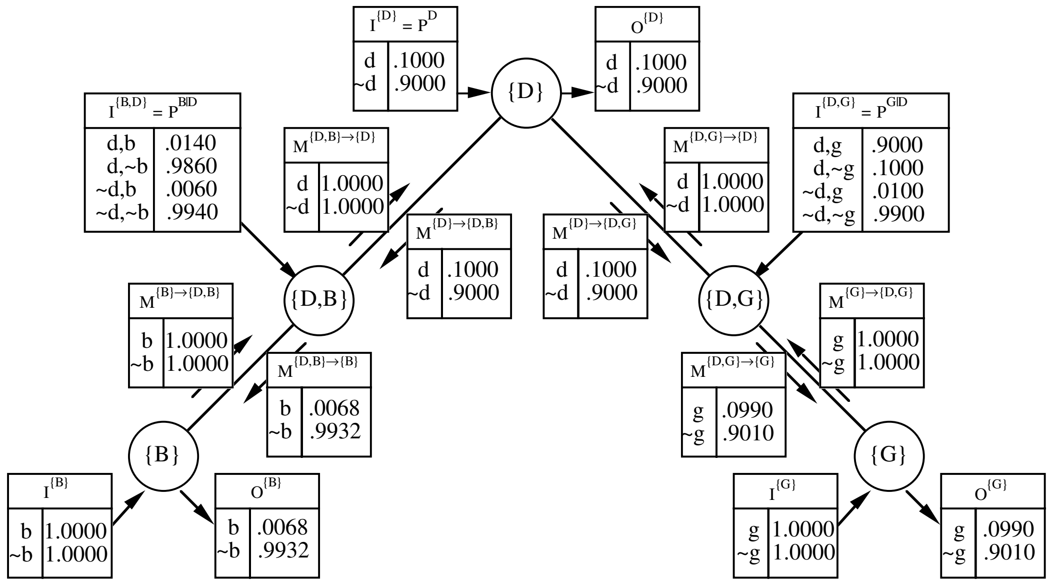
The enlarged hypertree and a Markov tree representation are shown in figure 8.

Figure 8. The hypertree and a Markov tree representation.



Suppose we propagate the potentials using the scheme described in section 8. The results are as shown in figure 9. For each vertex h, the input potentials are shown as I^h and the output potentials are shown as O^h. All the messages are also shown. Note that the output potentials have been normalized so that they represent marginal posterior probabilities.

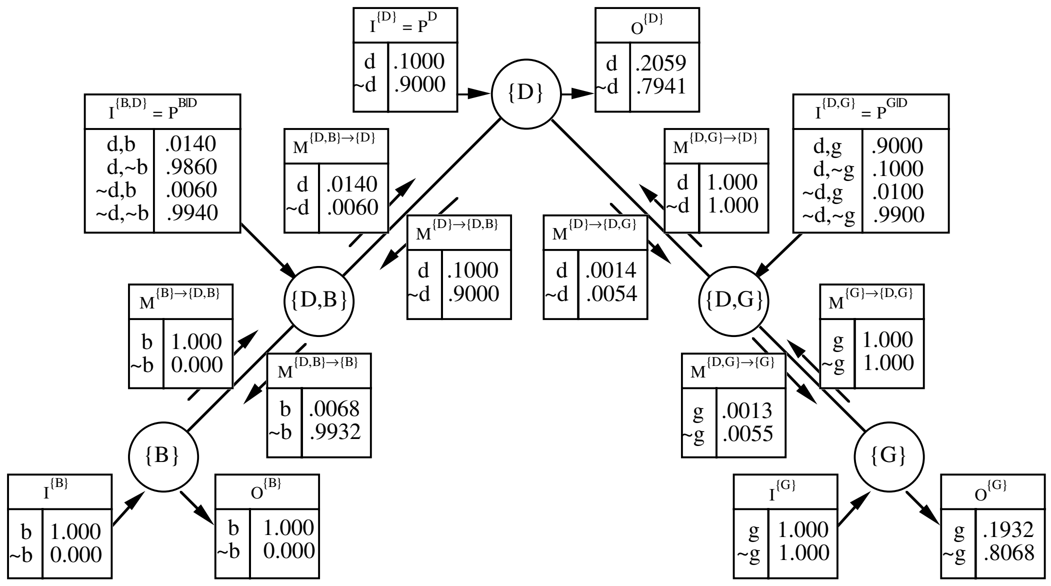
Figure 9. The initial propagation of potentials.



$I^{\{D\}} = P^D$: d .1000, ~d .9000
{D}
$O^{\{D\}}$: d .1000, ~d .9000
$I^{\{B,D\}} = P^{B|D}$: d,b .0140; d,~b .9860; ~d,b .0060; ~d,~b .9940
$M^{\{D,B\}\to\{D\}}$: d 1.0000, ~d 1.0000
$M^{\{D\}\to\{D,B\}}$: d .1000, ~d .9000
$M^{\{D\}\to\{D,G\}}$: d .1000, ~d .9000
$M^{\{D,G\}\to\{D\}}$: d 1.0000, ~d 1.0000
$I^{\{D,G\}} = P^{G|D}$: d,g .9000; d,~g .1000; ~d,g .0100; ~d,~g .9900
{D,B}
{D,G}
$M^{\{B\}\to\{D,B\}}$: b 1.0000, ~b 1.0000
$M^{\{D,B\}\to\{B\}}$: b .0068, ~b .9932
$M^{\{G\}\to\{D,G\}}$: g 1.0000, ~g 1.0000
$M^{\{D,G\}\to\{G\}}$: g .0990, ~g .9010
{B}
{G}
$I^{\{B\}}$: b 1.0000, ~b 1.0000
$O^{\{B\}}$: b .0068, ~b .9932
$I^{\{G\}}$: g 1.0000, ~g 1.0000
$O^{\{G\}}$: g .0990, ~g .9010

Now suppose we observe that the patient has blue toe. This is represented by the indicator potential for B=b. The other potentials are the same as before. If we propagate the potentials, the results are as shown in figure 10.

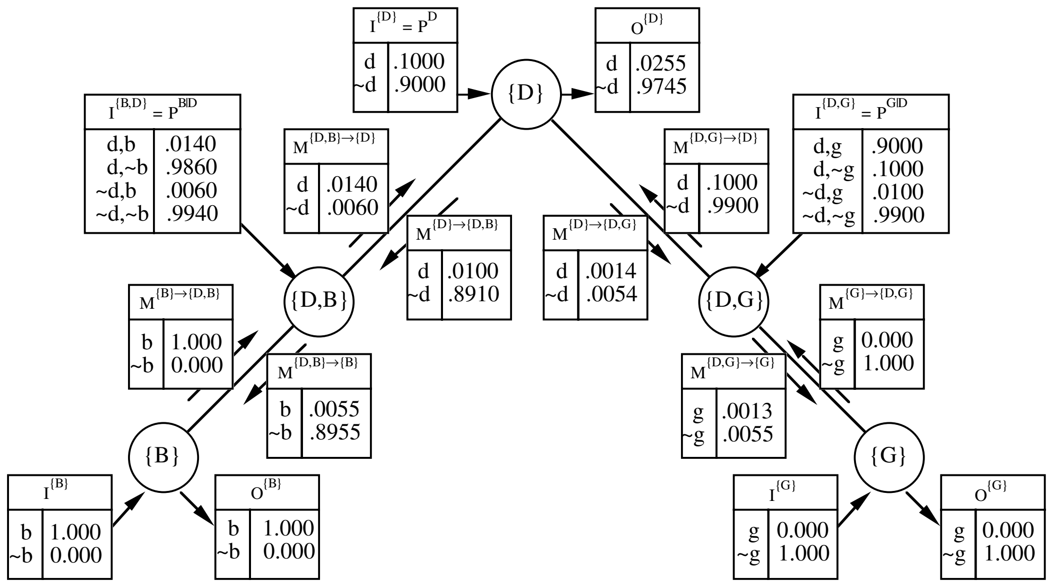
Figure 10. The results of propagation after the presence of blue toe is observed.



$I^{\{D\}} = P^D$: d .1000, ~d .9000
{D}
$O^{\{D\}}$: d .2059, ~d .7941
$I^{\{B,D\}} = P^{B|D}$: d,b .0140; d,~b .9860; ~d,b .0060; ~d,~b .9940
$M^{\{D,B\}\to\{D\}}$: d .0140, ~d .0060
$M^{\{D\}\to\{D,B\}}$: d .1000, ~d .9000
$M^{\{D\}\to\{D,G\}}$: d .0014, ~d .0054
$M^{\{D,G\}\to\{D\}}$: d 1.000, ~d 1.000
$I^{\{D,G\}} = P^{G|D}$: d,g .9000; d,~g .1000; ~d,g .0100; ~d,~g .9900
{D,B}
{D,G}
$M^{\{B\}\to\{D,B\}}$: b 1.000, ~b 0.000
$M^{\{D,B\}\to\{B\}}$: b .0068, ~b .9932
$M^{\{G\}\to\{D,G\}}$: g 1.000, ~g 1.000
$M^{\{D,G\}\to\{G\}}$: g .0013, ~g .0055
{B}
{G}
$I^{\{B\}}$: b 1.000, ~b 0.000
$O^{\{B\}}$: b 1.000, ~b 0.000
$I^{\{G\}}$: g 1.000, ~g 1.000
$O^{\{G\}}$: g .1932, ~g .8068

Note that the posterior probability of the presence of diabetes has increased (from .1 to .2059) and consequently the presence of glucose in urine has also increased (from .0990 to .1932). Now suppose that after the patient is tested for glucose in urine, the results indicate that there is an absence of glucose in urine. This information is represented by the indicator potential for G=~g. The other potentials are as before. If we propagate the potentials, the results are as shown in figure 11.

Figure 11. The results of propagation after the observation that patient does not have glucose in urine.



Note that the posterior probability of the presence of diabetes has decreased (from .2059 to .0255). This concludes our example.

Acknowledgments

Research for this article has been partially supported by NSF grant IRI-8610293. The authors have profited from conversation and correspondence with Chien-Chung Chan, Arthur Dempster, Yen-Teh Hsia, Augustine Kong, Khaled Mellouli, Judea Pearl, Debra Zarley, and Lianwen Zhang.

References

[1] S. Arnborg, D. G. Corneil and A. Proskurowski, Complexity of finding embeddings in a k-tree, *SIAM Journal of Algebraic and Discrete Methods*, 8(1987), 277-284.

[2] C. Beeri, R. Fagin, D. Maier and M. Yannakakis, On the desirability of acyclic database schemes, *Journal of the ACM*, 30(1983), 479-513.

[3] C. Berge, *Graphs and Hypergraphs*, translated from French by E. Minieka (North-Holland, 1973).

[4] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming* (Academic Press, 1972).

[5] L. S. Brownston, R. G. Farrell, E. Kant and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming* (Addison-Wesley, 1985).

[6] C. Cannings, E. A. Thompson and M. H. Skolnick, Probability functions on complex pedigrees, *Advances in Applied Probability*, 10(1978), 26-61.

[7] R. Dechter and J. Pearl, Tree-clustering schemes for constraint processing, in: *Proc. of the 7th National Conference on AI (AAAI-88)* (St. Paul, MN, 1988), 1, 150-154.

[8] A. P. Dempster, Construction and local computation aspects of network belief functions, Research Report S-125, Department of Statistics, Harvard University, Cambridge, MA (1988).

[9] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs* (Academic Press, 1980).

[10] D. Hunter, Parallel belief revision, in: *Proc. of the 4th Workshop on Uncertainty in AI* (Minneapolis, MN, 1988), 170-176.

[11] C. W. Kelly III and S. Barclay, A general Bayesian model for hierarchical inference, *Organizational Behavior and Human Performance*, 10(1973), 388-403.

[12] A. Kong, Multivariate belief functions and graphical models, Ph.D. dissertation, Department of Statistics, Harvard University, Cambridge, MA (1986).

[13] S. L. Lauritzen, T. P. Speed and K. Vijayan, Decomposable graphs and hypergraphs, *Journal of the Australian Mathematical Society*, series A, 36(1984), 12-29.

[14] S. L. Lauritzen and D. J. Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems (with discussion), *Journal of the Royal Statistical Society*, series B, 50(1988), 157-224.

[15] D. Maier, *The Theory of Relational Databases* (Computer Science Press, 1983).

[16] F. M. Malvestuto, Decomposing complex contingency tables to reduce storage requirements, in: *Proc. of the 1986 Conference on Computational Statistics* (1986), 66-71.

[17] R. J. Meinhold and N. D. Singpurwalla, Understanding the Kalman filter, *American Statistician*, 37(1983), 241-288.

[18] K. Mellouli, On the propagation of beliefs in networks using the Dempster-Shafer theory of evidence, Ph.D. dissertation, School of Business, University of Kansas, Lawrence, KS (1987).

[19] J. Pearl, Fusion, propagation and structuring in belief networks, *Artificial Intelligence*, 29(1986), 241-288.

[20] D. J. Rose, Triangulated graphs and the elimination process, *Journal of Mathematical Analysis and Applications*, 32(1970), 597-609.

[21] D. J. Rose, A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in: *Graph Theory and Computing*, ed. R.C. Read (Academic Press, 1973), 183-217.

[22] R. Seidel, A new method for solving constraint satisfaction problems, in: *Proc. of the 7th International Joint Conference on AI (IJCAI-81)* (Vancouver, British Columbia, Canada, 1981), 1, 338-342.

[23] R. D. Shachter and D. Heckerman, A backwards view for assessment, *AI Magazine*, 8(1987), 55-61.

[24] G. R. Shafer and P. P. Shenoy, Local computation in hypertrees, Working Paper No. 201, School of Business, University of Kansas, Lawrence, KS (1988).

[25] G. Shafer, P. P. Shenoy and K. Mellouli, Propagating belief functions in qualitative Markov trees, *International Journal of Approximate Reasoning*, 1(1987), 349-400.

[26] P. P. Shenoy, A Valuation-based language for expert systems, *International Journal of Approximate Reasoning* (1989), to appear.

[27] P. P. Shenoy and G. Shafer, Propagating belief functions using local computations, *IEEE Expert*, 1(1986), 43-52.

[28] P. P. Shenoy and G. Shafer, An axiomatic framework for Bayesian and belief-function propagation, in: *Proc. of the 4th Workshop on Uncertainty in Artificial Intelligence* (St. Paul, MN, 1988), 307-314.

[29] P. P. Shenoy and G. R. Shafer, Constraint propagation, Working Paper No. 208, School of Business, University of Kansas, Lawrence, KS (1988).

[30] P. P. Shenoy and G. R. Shafer, Axioms for discrete optimization using local computation, Working Paper No. 207, School of Business, University of Kansas, Lawrence, KS (1988).

[31] P. P. Shenoy, G. Shafer and K. Mellouli, Propagation of belief functions: A distributed approach, in: *Proc. of the 2nd Workshop on Uncertainty in AI* (Philadelphia, PA, 1986), 249-260, 1986. Also in: *Uncertainty in Artificial Intelligence 2*, eds. J. F. Lemmer and L. N. Kanal (North-Holland, 1988), 325-335.

[32] W. Spohn, Ordinal conditional functions: A dynamic theory of epistemic states, in: *Causation in Decision, Belief Change, and Statistics*, eds. W. L. Harper and B. Skyrms (D. Reidel Publishing Company, 1988), II, 105-134.

[33] R. E. Tarjan and M. Yannakakis, Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, *SIAM Journal of Computing*, 13(1984), 566-579.

[34] L. Zhang, Studies on finding hypertree covers for hypergraphs, Working Paper No. 198, School of Business, University of Kansas, Lawrence, KS (1988).