Axioms for Probability and Belief-Function Propagation

Prakash P. Shenoy and Glenn Shafer

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

In this paper, we describe an abstract framework and axioms under which exact local computation of marginals is possible. The primitive objects of the framework are variables and valuations. The primitive operators of the framework are combination and marginalization. These operate on valuations. We state three axioms for these operators and we derive the possibility of local computation from the axioms. Next, we describe a propagation scheme for computing marginals of a valuation when we have a factorization of the valuation on a hypertree. Finally we show how the problem of computing marginals of joint probability distributions and joint belief functions fits the general framework.

1. INTRODUCTION

In this paper, we describe an abstract framework and present axioms for local computation of marginals in hypertrees. These axioms justify the use of local computation to find marginals for a probability distribution or belief function when the probability distribution or belief function is factored on a hypertree. The axioms are abstracted from the belief-function work of the authors (e.g., Shenoy and Shafer [1986], Shenoy et al [1988], Shafer et al [1987]), but they apply to probabilities as well as to belief functions.

In the probability case, the factorization is usually a factorization of a joint probability distribution, perhaps into marginals and conditionals. Probability factorizations sometimes arise from causal models, which relate each variable to a relatively small number of immediate causes; see e.g., Pearl [1986]. Probability factorizations can also arise from statistical models; see e.g., Darroch et al [1980]. Belief-function factorizations generally arise from the decomposition of evidence into independent items, each involving only a few

variables. We represent each item of evidence by a belief function and combine these belief functions by Dempster's rule [Shafer 1976].

It is shown in Shenoy [1989b] that Spohn's [1988, 1990] theory of epistemic beliefs also fits in the abstract framework described here. This framework is extended in Shenoy and Shafer [1988a,b] to include constraint propagation and optimization using local computation.

We first present our general axiomatic framework and then explain how it applies to probabilities and belief functions. Before we can present the axiomatic framework, we need to review some graph-theoretic concepts. We do this in section 2. We present the framework in section 3. We apply it to probabilities in section 4 and to belief functions in section 5.

2. SOME CONCEPTS FROM GRAPH THEORY

Most of the concepts reviewed here have been studied extensively in the graph theory literature (see Berge [1973], Golumbic [1980], and Maier [1983]). A number of 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 [1983]) or decomposable hypergraph (Lauritzen et al [1984]). A *construction sequence* is what other authors have called a sequence with the running intersection property (Lauritzen and Spiegelhalter [1988]). A *Markov tree* is what authors in database theory have called a join tree (see Maier [1983]). 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. For a fuller explanation of the concepts reviewed here, see Shafer and Shenoy [1988].

As we shall see, hypertrees are closely related to Markov trees. The vertices of a Markov tree are always hyperedges of a hypertree, and the hyperedges of a hypertree can always be arranged in a Markov tree.

Hypergraphs and Hypertrees. We call a nonempty set $\mathcal{H}$ of nonempty 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 $\mathcal{H}$, $t \cap b \neq \emptyset$, and b contains every vertex of t that is contained in a hyperedge of $\mathcal{H}$ other than t; if $X \in t$ and $X \in h$, where $h \in \mathcal{H}$ and $h \neq t$, then $X \in b$. 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.

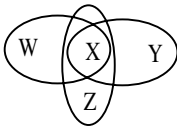
We call a hypergraph a *hypertree* if there is an ordering of its hyperedges, say $h_1 h_2 ... h_n$, such that h_k is a twig in the hypergraph $\{h_1, h_2, ..., h_k\}$ whenever $2 \leq k \leq n$. We call any such ordering of the hyperedges a *hypertree construction*

sequence for the hypertree. We call the first hyperedge in a hypertree construction sequence the *root* of the hypertree construction sequence.

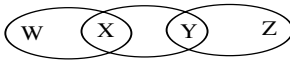
Figure 2.1 illustrates hypergraphs, hypertrees, twigs and construction sequences.

Figure 2.1. Some hypergraphs on {W,X,Y,Z}. The hypergraph $\mathcal{H}_1$ is a hypertree, all of its hyperedges are twigs, and all six orderings of its hyperedges are hypertree construction sequences. The hypergraph $\mathcal{H}_2$ is a hypertree, hyperedges {W,X} and {Y,Z} are twigs, and there are only four hypertree construction sequences: {W,X}{X,Y} {Y,Z}, {X,Y}{W,X}{Y,Z}, {X,Y}{Y,Z}{W,X}, and {Y,Z}{X,Y} {W,X}. The hypergraph $\mathcal{H}_3$ is not a hypertree and it has no twigs.

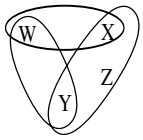
$\mathcal{H}_1$:



$\mathcal{H}_2$:



$\mathcal{H}_3$:



If we construct a hypertree by adding hyperedges following a hypertree construction sequence, then each hyperedge we add is a twig when it is added, and 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 at least one construction sequence beginning with that hyperedge.

Hypertree Covers of Hypergraphs. We will justify local computation under two assumptions. The joint probability distribution function or the joint belief

function 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.

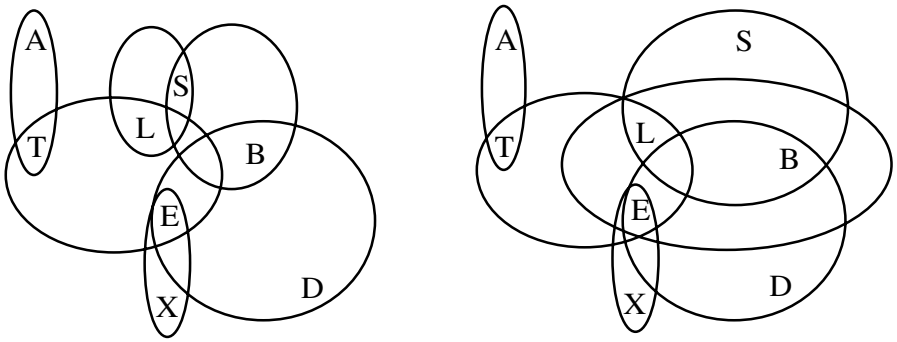
If the sets of variables form, instead, a hypergraph that is not a hypertree, then we must enlarge it until it is a hypertree. We can talk about this enlargement in two different ways. We can say we are adding larger hyperedges, keeping the hyperedges already there. Or, alternatively, we can say we are replacing the hyperedges already there with larger hyperedges. The choice between these two ways of talking matters little, 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 [1986], Mellouli [1987]).

Formally, 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 $h^* \supseteq h$. 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 2.2 shows a hypergraph that is not a hypertree and a hypertree cover for it.

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 $\mathcal{X}$. Finding a hypertree cover without large hyperedges, or finding a hypertree cover whose largest hyperedge is as small as possible, may be very difficult. How to do this best is the subject of a growing literature; see e.g., Rose [1970], Bertele and Brioschi [1972], Tarjan and Yannakakis [1984], Kong [1986], Arnborg et al [1987], Mellouli [1987], and Zhang [1988].

Trees. A *graph* is a pair $(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is a nonempty 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*.

Figure 2.2. *Left:* A hypergraph that is not a hypertree. *Right*: A hypertree cover for it obtained by adding hyperedges {S,L,B} and {L,E,B} and removing hyperedges {S,L} and {S,B}.



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 $(\{v_1,v_2,...,v_k\}, \mathcal{E}_k)$ whenever $2\le k\le n$, 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. Note that in a tree, for any two distinct vertices v_i and v_j, there is a unique path from v_i to v_j.

If we construct a tree following a tree construction sequence starting with the root and adding vertices, then each vertex we add is a leaf when it is added, and 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 have just defined a tree as a pair $(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is the set of vertices, and $\mathcal{E}$ is the set of edges. In the case of a Markov tree, the vertices are themselves nonempty sets. In other words, the set $\mathcal{V}$ is a hypergraph. In fact, it turns out to be a hypertree.

Here is our full definition. 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 $h\cap h'\neq\emptyset$.

(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 $\mathcal{E} = \{(h_2,\beta(h_2)),\ldots,(h_n,\beta(h_n))\}$; $h_1h_2...h_n$ is a tree construction sequence for $(\mathcal{H},\mathcal{E})$, and β is its budding.

See Shafer and Shenoy [1988] for a proof of Proposition 1. The key point here is the fact that a leaf in the Markov tree is a twig in the hypertree. This means that as we delete leaves from a Markov tree (a visually transparent operation), we are deleting twigs from the hypertree.

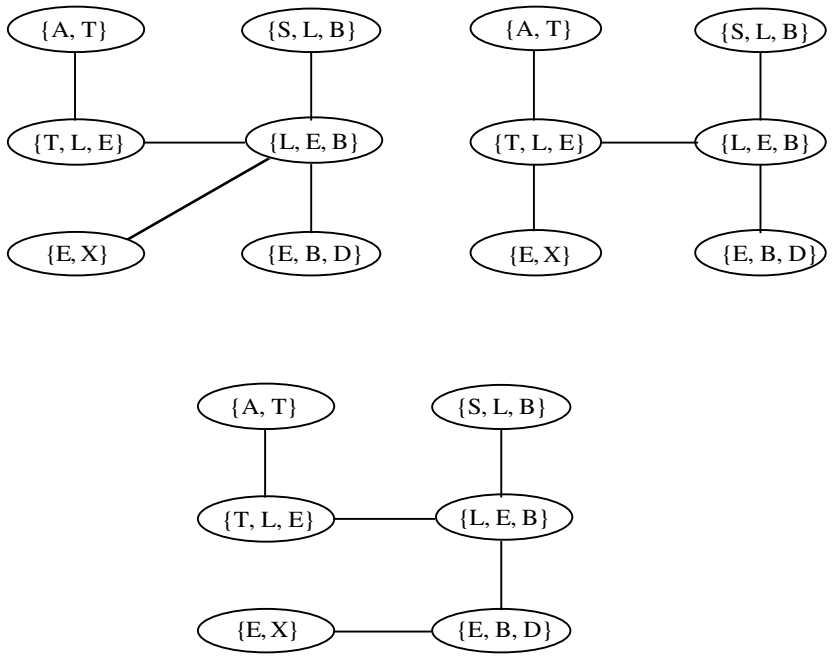
If $(\mathcal{H},\mathcal{E})$ is a Markov tree, then we call $(\mathcal{H},\mathcal{E})$ a *Markov tree representative* for the hypertree $\mathcal{H}$. As per Proposition 1, every hypertree has a Markov tree representative. Most hypertrees have more than one. Figure 2.3 shows three Markov tree representations for the hypertree in Figure 2.2.

3. AXIOMS FOR LOCAL COMPUTATION

In this section, we describe a set of axioms under which exact local computation of marginals is possible.

In section 3.1, we describe the framework for the axioms. The primitive objects of the framework are variables and valuations. The framework has two primitive operators, combination and marginalization. These operate on valuations. We state three axioms for these operators.

Figure 2.3. If we choose {L,E,B} as the root for the hypertree in Figure 2.2, then {L,E,B} must serve as the branch for {T,L,E}, {E,B,D}, and {S,L,B}, and {T,L,E} must serve as the branch for {A,T}. This leaves only {E,X}, which can use {L,E,B}, {T,L,E}, or {E,B,D} as its branch. It follows that the hypertree has exactly three Markov tree representations, which differ only in where the leaf {E,X} is attached.



In section 3.2, we show how local computation can be used to marginalize a factorization (of a valuation) 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 3.3, we shift our attention from a 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 valuations for the variables it represents and then send the result to a neighboring processor. In section 3.4, we generalize this idea to

a scheme of simultaneous computation and message passing that produces marginals for all the vertices in the Markov tree.

3.1. The Axiomatic Framework

The primitive objects of the framework are a finite set of variables and a set of valuations. The framework has two primitive operators: combination and marginalization. These operate on valuations.

Variables and Valuations. Let $\mathcal{X}$ be a finite set. The elements of $\mathcal{X}$ are called *variables*. For each $h \subseteq \mathcal{X}$, there is a set $\mathcal{V}_h$. The elements of $\mathcal{V}_h$ are called *valuations on h*. Let $\mathcal{V}$ denote $\cup\{\mathcal{V}_h | h \subseteq \mathcal{X}\}$, the set of all valuations.

In the case of probabilities, a valuation on h will be a non-negative, real-valued function on the set of all configurations of h (a configuration of h is a vector of possible values of variables in h). In the belief-function case, a valuation is a non-negative, real-valued function on the set of all subsets of configurations of h.

Proper Valuations. For each $h \subseteq \mathcal{X}$, there is a subset $\mathcal{P}_h$ of $\mathcal{V}_h$ whose elements will be called *proper valuations on h*. Let $\mathcal{P}$ denote $\cup\{\mathcal{P}_h | h \subseteq \mathcal{X}\}$, the set of all proper valuations. The notion of proper valuations is important as it will enable us to define combinability of valuations.

In the probability case, a valuation H on h is said to be proper if the values of the function H are not zero for all configurations of h. In the belief function case, a valuation H on h is said to be proper if the values of the function H are not zero for all nonempty subsets of configurations of h.

Combination. We assume there is a mapping $\otimes: \mathcal{V} \times \mathcal{V} \to \mathcal{V}$, called *combination*, such that

(i) If G and H are valuations on g and h respectively, then G⊗H is a valuation on g∪h; and

(ii) If either G or H is not a proper valuation, then G⊗H is not a proper valuation;

(iii) If G and H are both proper valuations, then G⊗H may or may not be a proper valuation.

If G⊗H is not a proper valuation, then we shall say that G and H are *not combinable*. If G⊗H is a proper valuation, then we shall say that G and H are *combinable* and that G⊗H is the *combination of G and H*.

Intuitively, combination corresponds to aggregation. If G and H represent information about variables in g and h, respectively, then G⊗H represents the

aggregated information for variables in g∪h. In the probability case, combination corresponds to pointwise multiplication. In the belief function case, combination corresponds to Dempster's rule.

Marginalization. We assume that for each $h \subseteq \mathcal{X}$, there is a mapping $\downarrow h: \cup\{\mathcal{V}_g | g \supseteq h\} \rightarrow \mathcal{V}_h$, called *marginalization to h*, such that

(i) If G is a valuation on g and $h \subseteq g$, then $G^{\downarrow h}$ is a valuation on h;

(ii) If G is a proper valuation, then $G^{\downarrow h}$ is a proper valuation; and

(iii) If G is not a proper valuation, then $G^{\downarrow h}$ is not a proper valuation.

We call $G^{\downarrow h}$ *marginal of G for h*.

Intuitively, marginalization corresponds to narrowing the focus of a valuation. If G is a valuation on g representing some information about variables in g, and $h \subseteq g$, then $G^{\downarrow h}$ represents the information for variables in h implied by G if we disregard variables in g-h. In both the probability and belief-function cases, marginalization corresponds to summation.

The Problem. We are now in a position to describe the problem. Suppose $\mathcal{H}$ is a hypergraph on $\mathcal{X}$. For each $h \in \mathcal{H}$, we have a proper valuation A_h on h. First, we need to determine if the proper valuations in the set $\{A_h | h \in \mathcal{H}\}$ are combinable. If the answer is in the affirmative, then let A denote the proper valuation $\otimes\{A_h | h \in \mathcal{H}\}$. Second, we need to find the marginal of A for each $X \in \mathcal{X}$.

If $\mathcal{X}$ is a large set of variables, then computation of $A^{\downarrow\{X\}}$ by first computing the joint valuation A on $\mathcal{X}$ and then marginalizing A to {X} will not be possible. For example, if we have 50 variables and each variable has 2 possible values, then we will have 2^{50} possible configurations of $\mathcal{X}$. Thus in the probability case, computing A will involve finding 2^{50} values. And in the belief function case, computing A will involve finding $2^{(2^{50})}$ values. In either case, the task is infeasible. We will state axioms for combination and marginalization that make it possible to use local computation to determine if the given proper valuations are combinable and to compute $A^{\downarrow\{X\}}$ for each $X \in \mathcal{X}$ if they are.

We will assume that these two mappings satisfy three axioms.

Axiom A1 (*Commutativity and associativity of combination*): Suppose G, H, K are valuations on g, h, and k respectively. Then $G \otimes H = H \otimes G$, and $G \otimes (H \otimes K) = (G \otimes H) \otimes K$.

Axiom A2 (*Consonance of marginalization*): Suppose G is a valuation on g, and suppose $k \subseteq h \subseteq g$. Then $(G^{\downarrow h})^{\downarrow k} = G^{\downarrow k}$.

Axiom A3 (*Distributivity of marginalization over combination*): Suppose G and H are valuations on g and h, respectively. Then $(G \otimes H)^{\downarrow g} = G \otimes (H^{\downarrow g \cap h})$

One implication of Axiom A1 is that when we have multiple combinations of valuations, we can write it without using parenthesis. For example, $(...((A_{h_1} \otimes A_{h_2}) \otimes A_{h_3}) \otimes ... \otimes A_{h_n})$ can be written simply as $\otimes\{A_{h_i} | i=1,...,n\}$ without indicating the order in which the combinations are carried out.

Factorization. Suppose A is a valuation on a finite set of variables $\mathcal{X}$, and suppose $\mathcal{H}$ is a hypergraph on $\mathcal{X}$. If A is equal to the combination of valuations on the hyperedges of h, say $A = \otimes\{A_h | h \in \mathcal{H}\}$, where A_h is a valuation on h, then we say that A *factorizes on* $\mathcal{H}$.

If we regard marginalization as a reduction of a valuation by deleting variables, then axiom A2 can be interpreted as saying that the order in which the variables are deleted does not matter.

Axiom A3 is the crucial axiom that makes local computation possible. Axiom A3 states that computation of $(G \otimes H)^{\downarrow g}$ can be accomplished without having to compute $G \otimes H$.

3.2. 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 computations, computations involving only the valuations on the twig and a branch for the twig. This elimination of a twig by local computation is the key to the computation of marginals from a factorization on a hypertree, for by successively deleting twigs, we can reduce the hypertree to a single hyperedge.

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) = $\cup(\mathcal{H}$-{t}).

Suppose A is a valuation on $\mathcal{X}$, suppose A factors on $\mathcal{H}$, and suppose we have stored A in a factored form. In other words, we have stored a valuation A_h for each h in $\mathcal{H}$, and we know that $A = \otimes\{A_h | h \in \mathcal{H}\}$. Adapting this factorization on A on $\mathcal{H}$ to the deletion of the twig t means reducing it to a factorization of $A^{\downarrow \mathcal{X}'}$ on $\mathcal{H}$-{t}. Can we do this? Yes. The following proposition tells us that if A factors on $\mathcal{H}$, then $A^{\downarrow \mathcal{X}'}$ factors on $\mathcal{H}$-{t}, and the second

factorization can be obtained from the first by a local computation that involves only t and a branch.

Proposition 2. Under the assumptions of the preceding paragraph,

$$A^{\downarrow \mathcal{X}} = (A_b \otimes A_t^{\downarrow t \cap b}) \otimes \big(\otimes\{A_h | h \in \mathcal{H}-\{t,b\}\}\big), \tag{3.1}$$

where b is any branch for t. Thus the marginal $A^{\downarrow \mathcal{X}}$ factors on the hypergraph $\mathcal{H}-\{t\}$. The valuation on b is combined with $A_t^{\downarrow t \cap b}$, and the valuations on the other elements of $\mathcal{H}-\{t\}$ are unchanged.

Proposition 2 follows directly from axiom A3 by letting $G = \otimes\{A_h | h \in \mathcal{H}-\{t\}\}$ and $H = A_t$.

This result is especially interesting in the case of hypertrees, because in this case repeated application of (3.1) 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 $h_1 h_2 ... h_n$. Suppose $\mathcal{X}_k$ denotes $h_1 \cup ... \cup h_k$ and $\mathcal{H}_k$ denotes $\{h_1, h_2, ..., h_k\}$ for k=1,...,n-1. We use (3.1) 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 (3.1) 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 $A^{\downarrow \mathcal{X}_1}$ on the hypertree $\mathcal{H}_1$ - 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 a branch. Note that such a step-wise computation of the marginal of A for h_1 is allowed by axiom A2.

3.3. 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 marginalized h_k's valuation to the intersection with its 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 valuation 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 marginal 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.

We cover a number of topics in this section. We begin by reviewing our marginalization scheme in the hypertree setting and seeing how its details translate into the Markov tree setting. We formulate precise descriptions of the operations that are carried out by each vertex and precise definitions of the messages that are passed from one vertex to another. Then we turn to questions of timing - whether a vertex uses a message as soon as it is received or waits for all its messages before it acts, how the order in which the vertices act are constrained, and whether the vertices act in serial or in parallel. We explain how the Markov tree can be expanded into an architecture for the parallel computation, with provision for storing messages as well as directing them. We explain how this architecture handles updating when inputs are changed. And finally, we explain how our computation can be directed by a simple forward-chaining production system.

Translating to the Markov Tree. We now translate our marginalization scheme from the hypertree to the Markov tree.

Recall the details in the hypertree setting. We have a valuation A on $\mathcal{X}$, in the form of a factorization on a hypertree $\mathcal{H}$. We want the marginal for the hyperedge h_1. We choose a hypertree construction sequence with h_1 as its root, say $h_1h_2...h_n$, and we choose a branching β for $h_1h_2...h_n$. On each hyperedge h_i, we have a valuation A_{h_i}. We repeatedly apply the following operation:

> *Operation H.* Marginalize the valuation now on h_k to $\beta(h_k)$. Change the valuation now on $\beta(h_k)$ by combining it by this marginal.

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

We want now to redescribe Operation H, and the process of its repeated application, in terms of the actions of processors located at the vertices of 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 of the h_i. The processor attached to h_i can store a valuation defined on h_i, can compute the marginal of this valuation to h_j, where h_j is a neighboring vertex, can send the marginal to h_j as a message, can accept a valuation on h_i as a message from a neighbor, and can change the valuation it has stored by combining 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 this 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.

In order to recast Operation H in terms of these processors, we need some more notation. Let Cur_h denote the valuation 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 $Cur_h^{\downarrow h\cap\beta(h)}$, the marginal of Cur_h to $\beta(h)$. It sends $Cur_h^{\downarrow h\cap\beta(h)}$ as a message to vertex $\beta(h)$. Vertex $\beta(h)$ accepts the message $Cur_h^{\downarrow h\cap\beta(h)}$ and changes $Cur_{\beta(h)}$ by multiplying it by $Cur_h^{\downarrow h\cap\beta(h)}$.

At the outset, $Cur_h = A_h$ 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 valuation Cur_{h_1}, the valuation 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. Since it is executed by a single processor, Operation M_2 will be easier for us to think about when we discuss alternative control regimes for the process of propagation.

Operation M_2 differs from Operation M_1 only in that it requires a processor to combine the messages it receives all at once, rather than incorporating them into the combination one by one as they arrive. Each time the Operation M_1 is executed for an h such that $\beta(h)=g$, the processor g must change the valuation it stores by combining it by the incoming message. But if processor g can store all its incoming messages, then it can delay the combination until it is its turn to marginalize. If we take this approach, then we can replace Operation M_1 with the following:

Operation M_{2a}. Vertex h combines the valuation A_h with all the messages it has received, and it calls the result Cur_h. Then it computes $Cur_h^{\downarrow h\cap\beta(h)}$, the marginal 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 combination, 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 combination 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 combine all its messages and its original valuation A_{h_1}, thus obtaining the marginal $A^{\downarrow h_1}$. We may call this Operation M_{2b}:

Operation M_{2b}. Vertex h combines the valuation A_h with all the messages it has received, and it reports the result to the user of the system.

So Operation M_2 actually consists of two operations. Operation M_{2a} is executed successively by h_n, h_{n-1}, and so on down to h_2. Then Operation M_{2b} is executed by h_1.

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. The formula will not make much immediate contribution to our understanding, but it will serve as a useful reference in the next section, where we discuss how to extend our scheme for computing a single marginal to a scheme for computing all marginals.

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 $Cur_h = A_h\otimes(\otimes\{M^{g\to\beta(g)} | g\in \mathcal{H} \text{ and } \beta(g)=h\})$. Putting these two formulas together, we have

$$M^{h\to\beta(h)} = \left(A_h\otimes(\otimes\{M^{g\to\beta(g)} | g\in \mathcal{H} \text{ and } \beta(g)=h\})\right)^{\downarrow h\cap\beta(h)}. \qquad (3.2)$$

If h is a leaf, then there is no $g \in \mathcal{H}$ such that $h=\beta(g)$, and so (3.2) reduces to

$$M^{h\to\beta(h)} = A_h{}^{\downarrow h\cap\beta(h)}. \tag{3.3}$$

Formula (3.2) 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 (3.3) defines $M^{h\to\beta(h)}$ for the leaves; a further application of (3.2) 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:

$$A^{\downarrow h} = A_h\otimes(\otimes\{M^{g\to\beta(g)} | g\in \mathcal{H} \text{ and } \beta(g)=h\}). \tag{3.4}$$

Storing the Messages. If we want to think in terms of Operation M_2, then we must imagine that our processors have a way to store incoming messages.

Figure 3.1 depicts an architecture that provides for such storage. The figure shows a storage register at vertex g for each of g's neighbors. The registers for neighbors on the side of g away from the goal vertex are used to store incoming messages. The register for the neighbor in the direction of the goal vertex is used to store the vertex's outgoing message. The registers serve as communication links between neighbors; the outgoing register for one vertex being the incoming register for its neighbor in the direction of the goal vertex.

The message $M^{g\to\beta(g)}$, which vertex g stores in the register linking g to its bud, is a valuation on $g\cap\beta(g)$. It is the marginal for the bud of a valuation on g.

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 the operations in a Markov tree. Consider, for example, the Markov tree with 15 vertices in Figure 3.2. 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 3.1. A typical vertex processor g, with incoming messages from vertices f and e and outgoing message to h; here $g=\beta(f)=\beta(e)$ and $h=\beta(g)$.

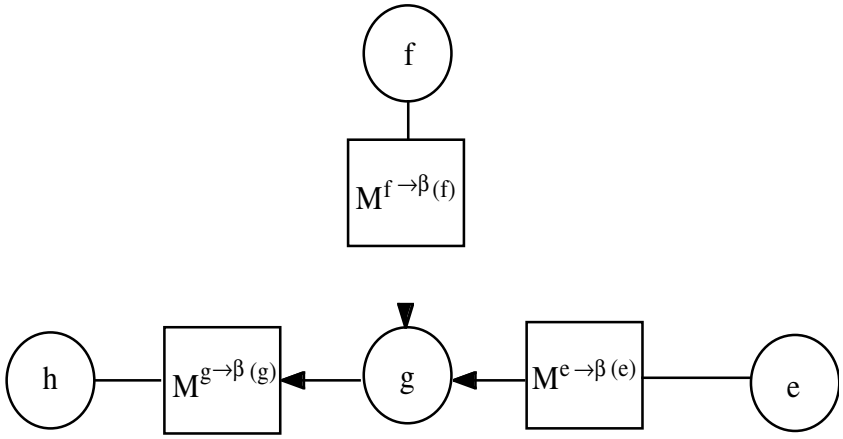
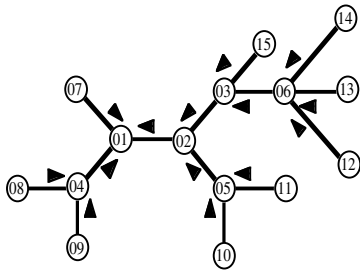


Figure 3.2. A tree with 15 vertices.



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 marginalization 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

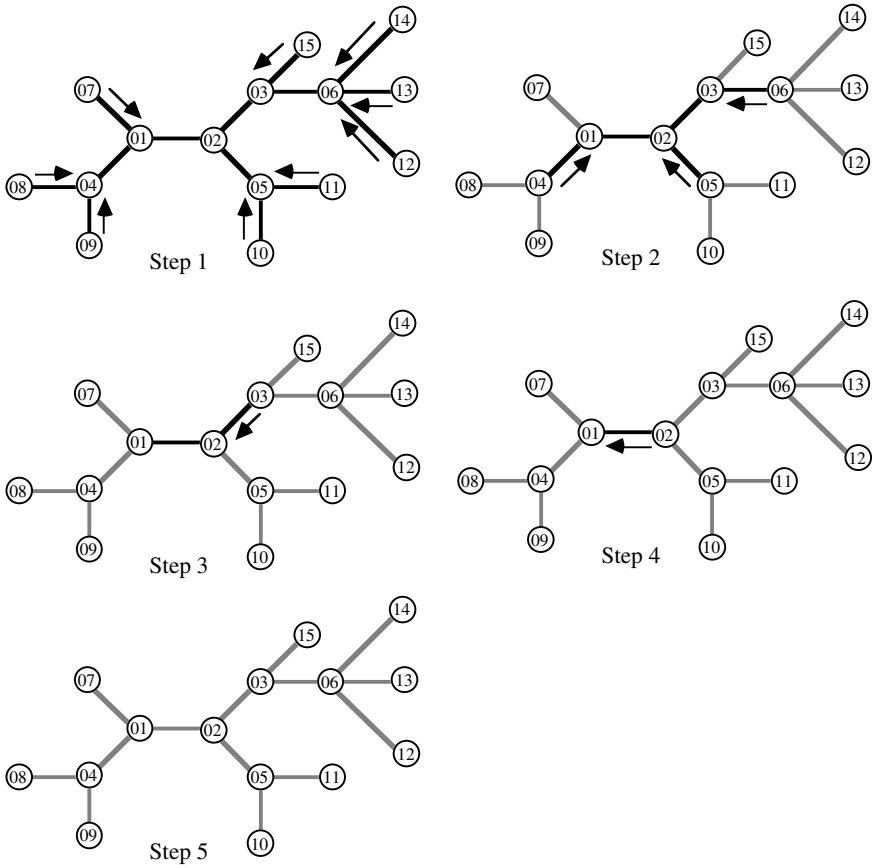
scheme on a serial computer. But if the different vertices really did have independent processors that could operate in parallel, then some of the vertices could act simultaneously. Figure 3.3 illustrates one way this might go for the Markov tree of Figure 3.2. In step 1, all the leaf processors project to their branches. In step 2, vertices 4, 5, and 6 (which would be leaves were the original leaves deleted) project. And so on.

If the different processors take different amounts of time to perform Operation M_2 on their inputs, then the lock-step timing of Figure 3.3 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. (In terms of Figure 3.1, each processor must wait until all its incoming registers are filled before it can compute a message to put in its outgoing register.) If 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 1986].

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 for their turn.

Figure 3.3. An example of the message-passing scheme for computation of the marginal of vertex 1.



*Updating Messages***.** Suppose we have completed the computation of $A^{\downarrow h_1}$, the marginal for our goal vertex. And suppose we now find reason to change A by changing 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 marginal $A^{\downarrow h_1}$ without discarding all the work we have already done. If we leave some of the inputs unchanged, then some of the computations may not need to be repeated.

Unnecessary computation can be avoided without global control. We simply need a way of marking valuations, 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 $M^{g\to\beta(g)}$ from a vertex g that has h as its bud), and can write the mark on its own output, the message $M^{h\to\beta(h)}$. When we wish to update the computation of $A^{\downarrow h_1}$, 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.

Of course, updating can also be achieved with much less control. As Pearl [1986] has emphasized, hardly any control at all is needed if we are indifferent to the possibility of wasted effort. If we do not care whether a processor repeats the same computations, we can forget about marking valuations and simply allow each processor to recompute its output from its inputs periodically or at random times. Under these circumstances, any change in one of the A_g will eventually be propagated through the system to change $A^{\downarrow h_1}$.

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 a 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. [1985] or Davis and King [1984].)

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 $M^{g\to\beta(g)}$ is in working memory for every g such that $\beta(g)=h$, then use (3.3) to compute $M^{h\to\beta(h)}$, and place it in working memory.
>
> ***Rule 2*:** If A_{h_1} is in working memory and $M^{g\to\beta(g)}$ is in working memory for every g such that $\beta(g)=h_1$, then use (3.4) to compute $A^{\downarrow h_1}$, and print the result.

Initially, there will be no $M^{g\to\beta(g)}$ 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. 1985, 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.

3.4. 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. As we will see, this can be done efficiently with only slight changes in the architecture or rules.

Computing all 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?

The first point to notice in answering this question is 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 3.4 to it to compute $A^{\downarrow h}$.

The second point to notice is 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 $M^{f\to\beta(f)}$ that we send from f to g when computing $A^{\downarrow h}$ is the same as the message $M^{f\to\beta'(f)}$ that we send from f to g when computing $A^{\downarrow h'}$. Since the value of $M^{f\to\beta(f)}$ does not depend on the budding β, we may write $M^{f\to g}$ instead of $M^{f\to\beta(f)}$ when $\beta(f)=g$.

If we compute marginals for all the vertices, then we will eventually compute both $M^{f\to g}$ and $M^{g\to f}$ for every edge $\{f,g\}$. We will compute $M^{f\to g}$ when we compute the marginal on g or on any other vertex on the g side of the edge, and we will compute $M^{g\to f}$ when we compute the marginal on g or on any other vertex on the g side of the edge.

We can easily generalize the recursive definition of $M^{g\to\beta(g)}$ that we gave in section 3.5 to a recursive definition of $M^{g\to h}$ for all neighbors g and h. To do so, we merely restate (3.2) in a way that replaces references to the budding β by references to neighbors and the direction of the message. We obtain

$$M^{g\to h} = \left(A_g \otimes (\otimes\{M^{f\to g} | f \in (\mathcal{N}_g - \{h\})\})\right)^{\downarrow g\cap h}, \tag{3.5}$$

where $\mathcal{N}_g$ is the set of all g's neighbors in $(\mathcal{H},\mathcal{E})$. If g is a leaf vertex, then (3.5) reduces to $M^{g\to h} = A_g{}^{\downarrow g\cap h}$.

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

$$A^{\downarrow h} = A_h \otimes (\otimes\{M^{g\to h} | g \in \mathcal{N}_h\}). \tag{3.6}$$

The General Architecture. A slight modification of the architecture shown in Figure 3.1 will allow us to implement the simultaneous computation of the marginals on all the hyperedges. We simply put two storage registers between every pair of neighbors f and g, as in Figure 3.4. One register stores the message from f to g; the other stores the message from g to f.

Figure 3.5 shows a more elaborate architecture for the simultaneous computation. In addition to the storage registers that communicate between vertices, this figure shows registers where the original valuations, the A_h, are put into the system and the marginals, the $A^{\downarrow h}$, are read out.

In the architecture of Figure 3.1, computation is controlled by the simple requirement that a vertex g must have messages in all its incoming registers before it can compute a message to place in its outgoing register. In the architecture of Figure 3.5, 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 Figure 3.1 carry over to Figure 3.5.

Figure 3.4. The two storage registers between f and g.

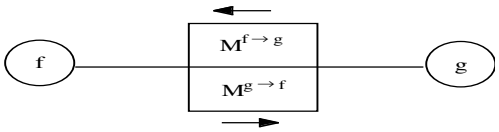
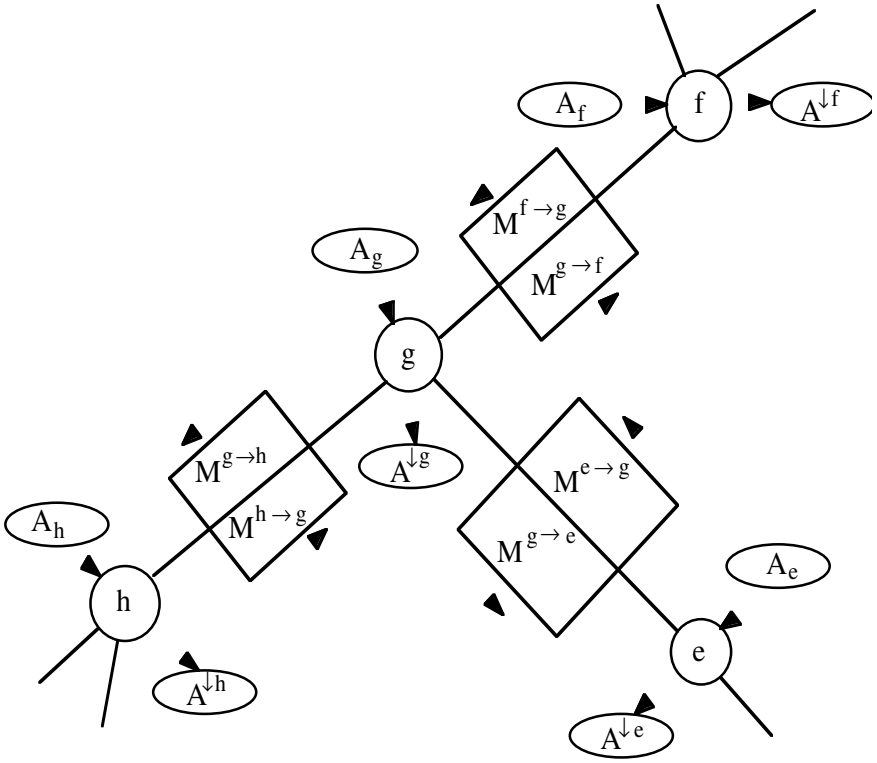


Figure 3.5. Several vertices, with storage registers for communication between themselves and with the user.



In particular, updating can be handled efficiently if a method is provided for marking updated inputs and messages. If we change just one of the input, 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 3.4. 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.

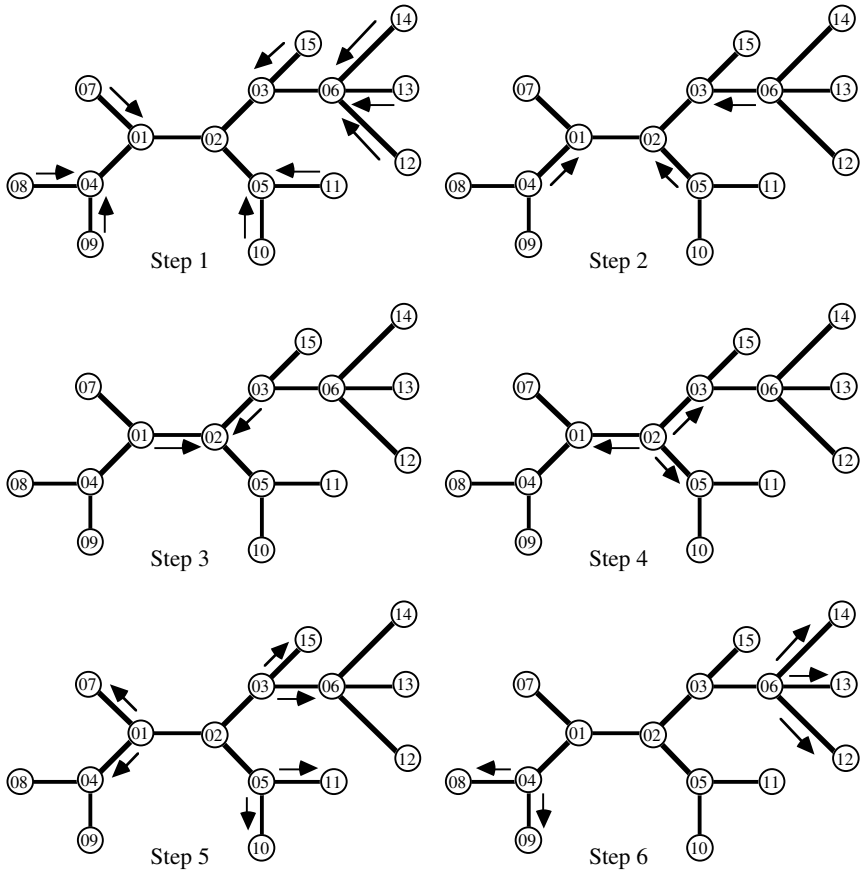
It may be enlightening to look at how the lock-step control we illustrated with Figure 3.3 might generalize to simultaneous computation of the marginals for all vertices. Consider a lock-step regime where at each step, each vertex looks and sees what messages it has the information to compute, computes 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 3.6 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.

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 3.6. 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. And whenever this number is even, there will 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, then we do not have to wait for all the computations that start together to finish before taking advantage of the ones that are finished to start new ones. We can allow a new computation to start whenever a processor is free and it has the information needed. On the other hand, we need not require that the work be done so promptly. We can assume that processors look for work to do only at random times. But no matter how we handle these issues, the computation will converge to some particular vertex or pair of neighboring vertices and then move back out from that vertex or pair of vertices.

Figure 3.6. An example of the message-passing scheme for simultaneous computation of all marginals.



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 from sending a message to any neighbor until it has received messages from all its neighbors. If we then let the system of Figure 3.6 run, it will behave just like the system of Figure 3.3 with h as the root, until h 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 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, and $M^{f\to g}$ is in working memory for every f in $\mathcal{N}_g$-{h}, then use (3.5) 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 (3.6) 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 3.5 three times.

4. PROBABILITY PROPAGATION

In this section, we explain local computation for probability distributions. More precisely, we show how the problem of computing marginals of joint probability distributions fits the general framework described in the previous section.

For probability propagation, proper valuations will correspond to potentials.

Potentials. 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*. We will be concerned with a finite set $\mathcal{X}$ of variables, and we will assume that all the variables in $\mathcal{X}$ have finite frames. For each $h\subseteq\mathcal{X}$, we let $\mathcal{W}_h$ denote the Cartesian product of $\mathcal{W}_X$ for X in h; $\mathcal{W}_h = \times\{\mathcal{W}_X | X\in h\}$. We call $\mathcal{W}_h$ the *frame for h*. We will refer to elements of $\mathcal{W}_h$ as *configurations of h*. A *potential on h* is a real-valued function on $\mathcal{W}_h$ that has non-negative values that are not all zero. Intuitively, potentials are unnormalized probability distributions.

Projection of configurations. In order to develop a notation for the combination of potentials, we first need a notation for the projection of configurations of a set of variables to a smaller set of variables. Here projection

simply 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 sets of variables, $h \subseteq g$, and $\mathbf{x}$ is a configuration of g, then we will let $\mathbf{x}^{\downarrow h}$ denote the projection of $\mathbf{x}$ to h.

Combination. For potentials, combination is simply pointwise multiplication. If G is a potential on g, H is a potential on h, and there exists an $\mathbf{x} \in \mathcal{W}_{g \cup h}$ such that

$$G(\mathbf{x}^{\downarrow g})H(\mathbf{x}^{\downarrow h}) > 0, \tag{4.1}$$

then their *combination*, denoted simply by GH, is the potential on $g \cup h$ given by

$$(GH)(\mathbf{x}) = G(\mathbf{x}^{\downarrow g})H(\mathbf{x}^{\downarrow h}) \tag{4.2}$$

for all $\mathbf{x} \in \mathcal{W}_{g \cup h}$. If there exists no $\mathbf{x} \in \mathcal{W}_{g \cup h}$ such that $G(\mathbf{x}^{\downarrow g})H(\mathbf{x}^{\downarrow h}) > 0$, then we say that G and H are *not combinable*.

Intuitively, if the bodies of evidence on which G and H are based are independent, then $G \oplus H$ is supposed to represent the result of pooling these two bodies of evidence. Note that condition (4.1) ensures that GH as defined in (4.2) is a potential. If condition (4.1) does not hold, this means that the two bodies of evidence corresponding to G and H contradict each other completely and it is not possible to combine such evidence.

It is clear from the definition of combination of potentials that it is commutative and associative (axiom A1).

Marginalization. Marginalization is familiar in probability theory; it 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 a potential on g. The *marginal of G for h*, denoted by $G^{\downarrow h}$, is the potential on h defined by

$$G^{\downarrow h}(\mathbf{x}) = \begin{cases} \Sigma\{G(\mathbf{x},\mathbf{y}) | \mathbf{y} \in \mathcal{W}_{g-h}\} & \text{if h is a proper subset of g} \\ G(\mathbf{x}) & \text{if h=g} \end{cases}$$

for all $\mathbf{x} \in \mathcal{W}_h$.

It is obvious from the above definition that marginalization operation for potentials satisfies axiom A2.

Since multiplication distributes over addition, it is easy to show that combination and marginalization for potentials satisfy axiom A3. Thus all axioms are satisfied making local computation possible.

A number of authors who have studied local computation for probability, including Kelly and Barclay [1973], Cannings, Thompson and Skolnick [1978], Pearl [1986], Shenoy and Shafer [1986], and Lauritzen and Spiegelhalter [1988], have described schemes that are variations on the the basic scheme described in section 2. Most of these authors, however, have justified their schemes by emphasizing 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, at any stage, in terms of conditional probabilities. For more regarding this point, see Shafer and Shenoy [1988].

We would like to make two important observations for the case of probability propagation. First note that it is sufficient, in order for a potential A to factor on $\mathcal{H}$, that A be proportional to a product of arrays on the hyperedges. Indeed, if

$$A \propto \Pi\{A_h | h \in \mathcal{H}\},$$

where A_h is a potential on h, then a representation of the form $A = \Pi\{A_h | h \in \mathcal{H}\}$ can be obtained simply by incorporating the constant of proportionality into one of the A_h. In practice, we will postpone finding the constant of proportionality until we have marginalized A to a hyperedge using the scheme described in section 2.

The second observation relates to conditioning joint probability distributions. Suppose a probability distribution P represents our assessment of a given body of information, and we have been computing marginals of P from the factorization

$$P = \Pi\{A_h | h \in \mathcal{H}\} \tag{4.3}$$

where $\mathcal{H}$ is a hypertree on $\mathcal{X}$. Suppose we now observe the values of some of the variables in $\mathcal{X}$; say we observe $Y_1=y_1$, $Y_2=y_2$, and so on up to $Y_n=y_n$. We change our assessment from P to $P^{|f=\mathbf{y}}$ where $f = \{Y_1, ..., Y_n\}$, $\mathbf{y} = \{y_1, ..., y_n\}$, and $P^{|f=\mathbf{y}}$ denotes the joint probability distribution conditioned on the observations. Can we adapt (4.3) to a factorization of $P^{|f=\mathbf{y}}$? Yes, we can. More precisely, we can adapt (4.3) to a factorization of a potential proportional to $P^{|f=\mathbf{y}}$, and this, as we noted in our first observation, is good enough. The adaptation is simple. It follows from the definition of conditional probability that

$$P^{|f=y} \propto B^{Y_1=y_1} \ldots B^{Y_n=y_n} \Pi\{A_h | h \in \mathcal{H}\}$$

where $B^{Y_i=y_i}$ is the *indicator potential for* $Y_i=y_i$ on $\{Y_i\}$ defined by

$$B^{Y_i=y_i}(x) = \begin{cases} 0 & \text{if } x \neq y_i \\ 1 & \text{if } x = y_i \end{cases}$$

for all $x \in \mathcal{W}_{Y_i}$.

We will now illustrate our propagation scheme using a simple example.

An Example. This example is adapted from Shachter and Heckerman [1987]. 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.

$$P = P^D P^{B|D} P^{G|D} \tag{4.4}$$

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 expand (4.4) 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 4.1. The enlarged hypertree and a Markov tree representation are shown in Figure 4.1.

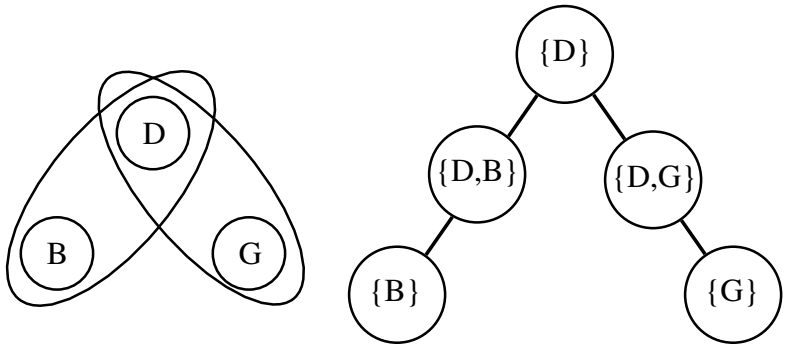
Table 4.1. The potentials P^D, $P^{B|D}$, and $P^{G|D}$.

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

Figure 4.1. The hypertree and a Markov tree representation.



Suppose we propagate the potentials using the scheme described in section 2. The results are as shown in Figure 4.2. 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.

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 4.3.

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 4.4.

Figure 4.2. The initial propagation of potentials.

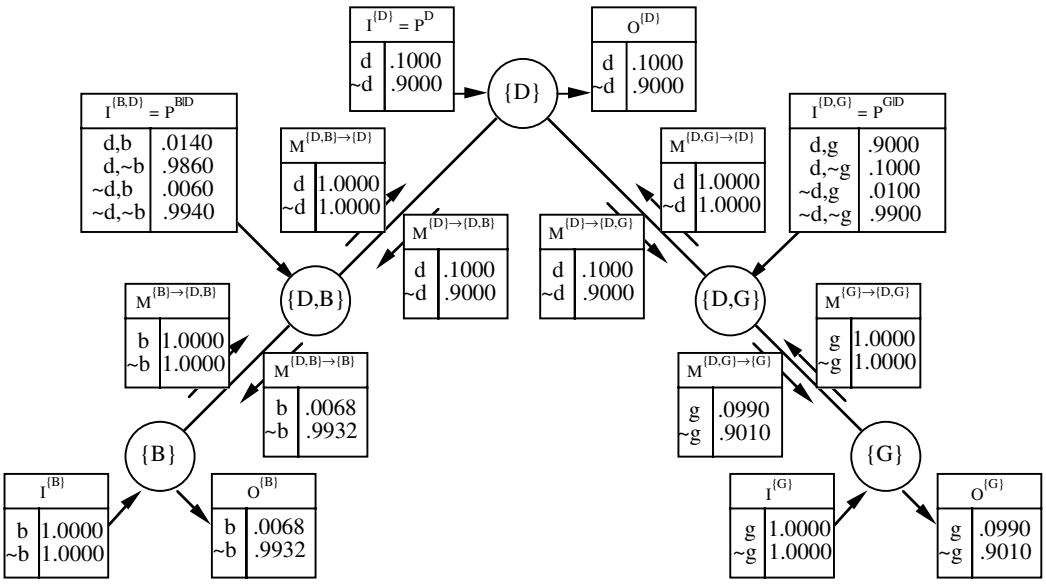


Figure 4.3. The results of propagation after the presence of blue toe is observed.

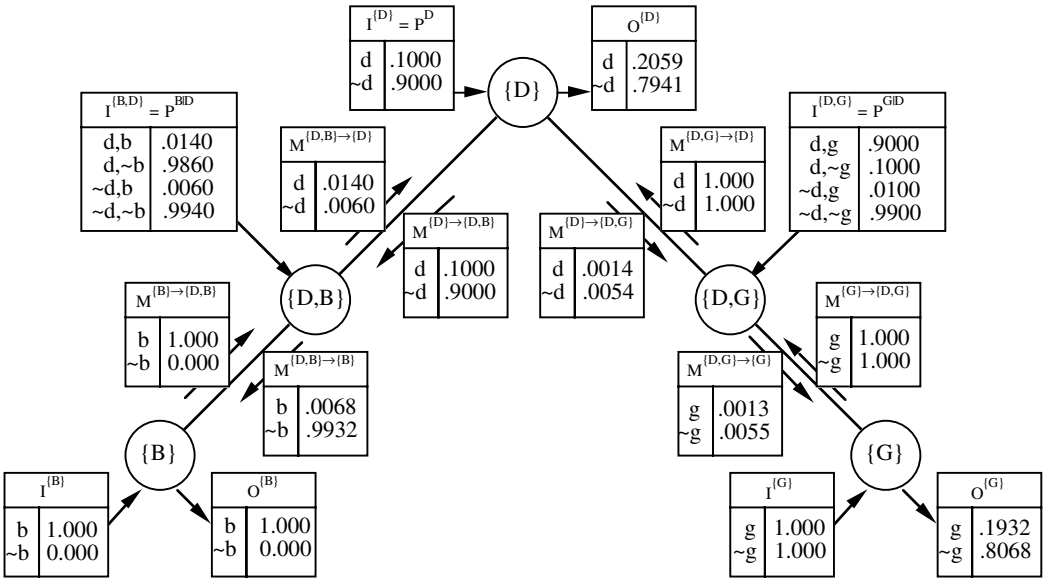
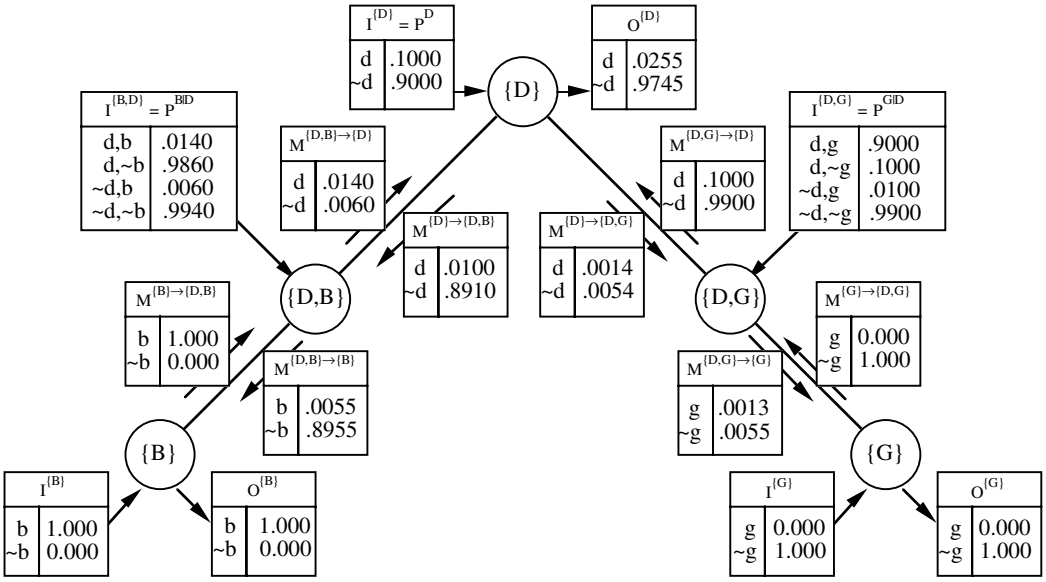


Figure 4.4. The results of propagation after the observation that patient does not have glucose in urine.



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

5. BELIEF-FUNCTION PROPAGATION

In this section, we explain local computation for belief functions. More precisely, we show how the problem of computing marginals of a joint belief function fits the general framework described in section 2.

For belief-function propagation, proper valuations correspond to either probability mass assignment functions, belief functions, plausibility functions or commonality functions. For simplicity of exposition, we will describe belief-function propagation in terms of superpotentials which are unnormalized basic probability assignment functions.

Basic Probability Assignment Functions. Suppose $\mathcal{W}_h$ is the frame for a subset h of variables. A *basic probability assignment function (bpa function) for h* is a non-negative, real-valued function M on the set of all subsets of $\mathcal{W}_h$ such that

(i) $M(\varnothing) = 0$, and

(ii) $\Sigma\{M(\alpha) | \alpha \subseteq \mathcal{W}_h\} = 1$.

Intuitively, M(a) represents the degree of belief assigned exactly to a (the proposition that the true configuration of h is in the set a) and to nothing smaller. A bpa function is the belief function equivalent of a probability mass assignment function in probability theory. Whereas a probability mass function is restricted to assigning probability masses only to singleton configurations of variables, a bpa function is allowed to assign probability masses to sets of configurations without assigning any mass to the individual configurations contained in the sets.

Superpotentials. Suppose h is a subset of variables. A *superpotential for h* is a non-negative, real-valued function on the set of all subsets of $\mathcal{W}_h$ such that the values of nonempty subsets are not all zero. Given a superpotential H on h, we can construct a bpa function H' for h from H as follows:

$$H'(\varnothing) = 0, \text{ and } H'(a) = H(a)/\Sigma\{H(b) | b \subseteq \mathcal{W}_h, b \neq \varnothing\}.$$

Thus superpotentials can be thought of as unnormalized bpa functions. Superpotentials correspond to the notion of proper valuations in the general framework.

Projection and Extension of Subsets. Before we can define combination and marginalization for superpotentials, we need the concepts of projection and extension of subsets of configurations.

If g and h are sets of variables, h⊆g, and g is a nonempty subset of $\mathcal{W}_g$, then the *projection of g to h*, denoted by $g^{\downarrow h}$, is the subset of $\mathcal{W}_h$ given by $g^{\downarrow h} = \{\mathbf{x}^{\downarrow h} | \mathbf{x} \in g\}$.

For example, If a is a subset of $\mathcal{W}_{\{W,X,Y,Z\}}$, then the marginal of a to {X,Y} consists of the elements of $\mathcal{W}_{\{X,Y\}}$ which can be obtained by projecting elements of a to $\mathcal{W}_{\{X,Y\}}$.

By extension of a subset of a frame to a subset of a larger frame, we mean a cylinder set extension. If g and h are sets of variables, h⊆g, h≠g, and h is a subset of $\mathcal{W}_h$, then the *extension of h to g* is $h \times \mathcal{W}_{g-h}$. If h is a subset of $\mathcal{W}_h$, then the extension of h to h is defined to be h. We will let $h^{\uparrow g}$ denote the extension of h to g.

For example, if a is a subset of $\mathcal{W}_{\{W,X\}}$, then the vacuous extension of a to {W,X,Y,Z} is $a \times \mathcal{W}_{\{Y,Z\}}$.

Combination. For superpotentials, combination is called Dempster's rule [Dempster 1966]. Consider two superpotentials G and H on g and h, respectively. If

$$\Sigma\{G(a)H(b) | (a^{\uparrow(g\cup h)}) \cap (b^{\uparrow(g\cup h)}) \neq \varnothing\} \neq 0, \qquad (5.1)$$

then their *combination*, denoted by G⊕H, is the superpotential on g∪h given by

$$G\oplus H(c) = \Sigma\{G(a)H(b) | (a^{\uparrow(g\cup h)}) \cap (b^{\uparrow(g\cup h)}) = c\} \qquad (5.2)$$

for all $c \subseteq \mathcal{W}_{g\cup h}$. If $\Sigma\{G(a)H(b) | (a^{\uparrow(g\cup h)}) \cap (b^{\uparrow(g\cup h)}) \neq \varnothing\} = 0$, then we say that G and H are *not combinable*.

Intuitively, if the bodies of evidence on which G and H are based are independent, then G⊕H is supposed to represent the result of pooling these two bodies of evidence. Note that condition (5.1) ensures that G⊕H defined in (5.2) is a superpotential. If condition (5.1) does not hold, this means that the two bodies of evidence corresponding to G and H contradict each other completely and it is not possible to combine such evidence.

It is shown in Shafer [1976] that Dempster's rule of combination is commutative and associative. Thus combination for superpotentials satisfies axiom A1.

Marginalization. Like marginalization for potentials, marginalization for superpotentials corresponds to summation.

Suppose G is a superpotential for g and suppose h⊆g. Then the *marginal of G for h* is the superpotential $G^{\downarrow h}$ for h defined as follows:

$$G^{\downarrow h}(a) = \Sigma\{G(b) | b \subseteq \mathcal{W}_g \text{ such that } b^{\downarrow h} = a\}$$

for all subsets a of $\mathcal{W}_h$.

It is easy to see that marginalization for superpotentials satisfies axiom A2. In Shafer and Shenoy [1988], it is shown that the above definitions of marginalization and combination for superpotentials satisfy axiom A3. Thus all axioms are satisfied making local computation possible.

Propagation of belief functions using local computation has been studied by Shafer and Logan [1987], Shenoy and Shafer [1986], Shenoy et al [1988], Kong [1986], Dempster and Kong [1986], Shafer et al [1987], Mellouli [1987], and Shafer and Shenoy [1988]. Shafer et al [1988], Shenoy [1989], Zarley [1988], Zarley et al [1988] and Hsia and Shenoy [1989, 1989b] discuss applications and implementations of these propagation schemes.

ACKNOWLEDGEMENTS

Research for this article has been partially supported by NSF grant IRI-8902444 and a Research Opportunities in Auditing grant 88-146 from the Peat Marwick Foundation. A condensed version appeared in the Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence in 1988.

REFERENCES

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

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

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

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

Buchanan, B. G. and Shortliffe, E. H., eds. (1984), *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley.

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

Darroch, J. N., Lauritzen, S. L. and Speed, T. P. (1980), Markov fields and log-linear models for contingency tables, *Annals of Statistics*, **8**, 522-539.

Davis, R. and King, J. J. (1984), The origin of rule-based systems in AI, in Buchanan and Shortliffe [1984, 20-52].

Dempster, A. P. (1966), New methods for reasoning toward posterior distributions based on sample data, *Annals of Mathematical Statistics*, **37**, 355-374.

Dempster, A. P. and Kong, A. (1986), Uncertain evidence and artificial analysis, Research Report S-108, Department of Statistics, Harvard University.

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

Hsia, Y. and Shenoy, P. P. (1989), An evidential language for expert systems, *Methodologies for Intelligent Systems, 4*, Ras, Z. (ed.), North-Holland, 9-16.

Hsia, Y. and Shenoy, P. P. (1989b), MacEvidence: A visual evidential language for knowledge-based systems, Working Paper No. 211, School of Business, University of Kansas.

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

Kong, A. (1986), Multivariate belief functions and graphical models, doctoral dissertation, Department of Statistics, Harvard University.

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

Lauritzen, S. L. and Spiegelhalter, D. J. (1988), Local computations with probabilities on graphical structures and their application to expert systems (with discussion), *Journal of the Royal Statistical Society*, series B, **50**(2), 157-224.

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

Mellouli, K. (1987), On the propagation of beliefs in networks using the Dempster-Shafer theory of evidence, doctoral dissertation, School of Business, University of Kansas.

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

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

Shachter, R. D. and Heckerman, D. (1987), A backwards view for assessment, *AI Magazine*, **8**(3), 55-61.

Shafer, G. (1976), *A Mathematical Theory of Evidence*, Princeton University Press.

Shafer, G. and Logan, R. (1987), Implementing Dempster's rule for hierarchical evidence, *Artificial Intelligence*, **33**, 271-298.

Shafer, G. and Shenoy, P. P. (1988), Local computation in hypertrees, Working Paper No. 201, School of Business, University of Kansas.

Shafer, G., Shenoy, P. P. and Mellouli, K. (1987), Propagating belief functions in qualitative Markov trees, *International Journal of Approximate Reasoning*, **1**(4), 349-400.

Shafer, G., Shenoy, P. P. and Srivastava, R. P. (1988), AUDITOR'S ASSISTANT: A knowledge engineering tool for audit decisions, *Auditing Symposium IX: Proceedings of the 1988 Touche Ross/University of Kansas Symposium on Auditing Problems*, 61-84.

Shenoy, P. P. (1989), A valuation-based language for expert systems, *International Journal of Approximate Reasoning*, **3**(5), 383-411.

Shenoy, P. P. (1989b), On Spohn's rule for revision of beliefs, Working Paper No. 213, School of Business, University of Kansas.

Shenoy, P. P. and Shafer, G. (1986), Propagating belief functions using local computations, *IEEE Expert*, **1**(3), 43-52.

Shenoy, P. P. and Shafer, G. (1988a), Axioms for discrete optimization using local computation, Working Paper No. 207, School of Business, University of Kansas.

Shenoy, P. P. and Shafer, G. (1988b), Constraint propagation, Working Paper No. 208, School of Business, University of Kansas.

Shenoy, P. P., Shafer, G. and Mellouli, K. (1988), Propagation of belief functions: A distributed approach, *Uncertainty in Artificial Intelligence 2*, Lemmer, J. F. and Kanal, L. N. (eds.), North-Holland, 325-336.

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

Spohn, W. (1990), A general non-probabilistic theory of inductive reasoning, this volume.

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

Zarley, D. K. (1988), An evidential reasoning system, Working Paper No. 206, School of Business, University of Kansas.

Zarley, D. K., Hsia, Y. T. and Shafer, G. (1988), Evidential reasoning using DELIEF, *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, **1**, 205-209, Minneapolis, MN.

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