A Differential Semantics for Jointree Algorithms

James D. Park

Computer Science Department
University of California
Los Angeles, CA 90095

Adnan Darwiche

Computer Science Department
University of California
Los Angeles, CA 90095

Abstract

A new approach to inference in belief networks has been recently proposed, which is based on an algebraic representation of belief networks using multi–linear functions. According to this approach, belief network inference reduces to a simple process of evaluating and differentiating multi–linear functions. We show here that mainstream inference algorithms based on jointrees are a special case of the approach based on multi–linear functions, in a very precise sense. We use this result to prove new properties of jointree algorithms. We also discuss some practical and theoretical implications of this new finding.

1 Introduction

It was recently shown that the probability distribution of a belief network can be represented using a multi–linear function, and that most probabilistic queries of interest can be retrieved directly from the partial derivatives of this function [3]. Although the multi–linear function has an exponential number

Email addresses: jd@cs.ucla.edu (James D. Park), darwiche@cs.ucla.edu (Adnan Darwiche).

of terms, it can be represented by a small arithmetic circuit in certain situations. For example, it was shown recently that real-world belief networks with treewidth up to 60 can be compiled into arithmetic circuits with few thousand nodes [4]. [1] Once a belief network is compiled into an arithmetic circuit, probabilistic inference is then performed by evaluating and differentiating the circuit, using a very simple procedure which resembles back–propagation in neural networks.

We show in this paper that mainstream inference algorithms based on jointrees [14,9] are a special-case of the arithmetic–circuit approach proposed in [3]. Specifically, we show that each jointree is an implicit representation of an arithmetic circuit; that the inward–pass in jointree propagation evaluates this circuit; that the outward–pass differentiates the circuit; and that the difference between Shenoy–Shafer and Hugin propagation is a difference between two numeric schemes for circuit differentiation. Using these results, we prove new useful properties of jointree propagation algorithms. We also suggest a new interpretation of the process of factoring graphical models into jointrees, as a process of factoring exponentially–sized multi–linear functions into arithmetic circuits of smaller size.

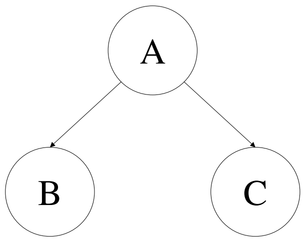
This paper is structured as follows. Sections 2 and 3 are dedicated to a review of inference approaches based on arithmetic circuits and jointrees. Section 4 shows that the jointree approach is a special case of the arithmetic–circuit approach, and discusses some practical implications of this finding. Section 5 discusses circuit differentiation in more details, explaining the difference between Shenoy–Shafer and Hugin propagation in those terms. Section 6 closes with a new perspective on factoring of graphical models based on these findings. Proofs of all theorems appear in Appendix A.

2 Belief networks as multi–linear functions

A belief network is a factored representation of a probability distribution. It consists of two parts: a directed acyclic graph (DAG) and a set of conditional probability tables (CPTs). For each node X and its parents $\mathbf{U}$ in the DAG, we must have a CPT that specifies the probability distribution of X under each instantiation $\mathbf{u}$ of the parents. [2] Figure 1 depicts a simple belief network which has three CPTs.

[1] Such networks have local structure, and are outside the scope of mainstream algorithms for inference in belief networks whose complexity is exponential in treewidth.

[2] Variables are denoted by upper–case letters (A) and their values by lower–case letters (a). Sets of variables are denoted by bold–face upper–case letters ($\mathbf{A}$) and their instantiations are denoted by bold–face lower–case letters ($\mathbf{a}$). For a variable A with values *true* and *false*, we use a to denote $A=$ *true* and $\bar{a}$ to denote $A=$ *false*.



A	B	
true	*true*	$\theta_{b\|a} = .2$
true	*false*	$\theta_{\bar{b}\|a} = .8$
false	*true*	$\theta_{b\|\bar{a}} = .7$
false	*false*	$\theta_{\bar{b}\|\bar{a}} = .3$

A	
true	$\theta_a = .6$
false	$\theta_{\bar{a}} = .4$

A	C	
true	*true*	$\theta_{c\|a} = .8$
true	*false*	$\theta_{\bar{c}\|a} = .2$
false	*true*	$\theta_{c\|\bar{a}} = .15$
false	*false*	$\theta_{\bar{c}\|\bar{a}} = .85$

Fig. 1. A belief network with its CPTs.

A belief network is a *representational* factorization of a probability distribution, not a *computational* one. That is, although the network compactly represents the distribution, it needs to be processed further if one is to obtain answers to arbitrary probabilistic queries. Mainstream algorithms for inference in belief networks operate on the network to generate a *computational* factorization, allowing one to answer queries in time which is linear in the factorization size. A most influential computational factorization of belief networks is the *jointree* [14,9,7]. Standard jointree factorizations are structure–based: their size depend only on the network topology and is invariant to local CPT structure. This observation has triggered much research for alternative, finer–grained factorizations, since real-world networks can exhibit significant local structure that leads to smaller factorizations if exploited.

We discuss next one of the latest proposals in this direction, which calls for using *arithmetic circuits* as a computational factorization of belief networks [3]. This proposal is based on viewing each belief network as a multi–linear function, which can be represented compactly using an arithmetic circuit. The multi–linear function itself contains two types of variables. *Evidence indicators:* for each variable X in the network , we have a variable λ_x for each value x of X. *Network parameters:* for each variable X and its parents $\mathbf{U}$ in the network, we have a variable $\theta_{x|\mathbf{u}}$ for each value x of X and instantiation $\mathbf{u}$ of $\mathbf{U}$.

The multi–linear function has a term for each instantiation of the network variables, which is constructed by multiplying all evidence indicators and network parameters that are consistent with that instantiation. For example, the

Finally, for a variable X and its parents $\mathbf{U}$, we use $\theta_{x|\mathbf{u}}$ to denote the CPT entry corresponding to $Pr(x \mid \mathbf{u})$.

multi–linear function of the network in Figure 1 has eight terms corresponding to the eight instantiations of variables A, B, C:

$$f = \lambda_a\lambda_b\lambda_c\theta_a\theta_{b|a}\theta_{c|a} + \lambda_a\lambda_b\lambda_{\bar{c}}\theta_a\theta_{b|a}\theta_{\bar{c}|a} + \ldots + \lambda_{\bar{a}}\lambda_{\bar{b}}\lambda_{\bar{c}}\theta_{\bar{a}}\theta_{\bar{b}|\bar{a}}\theta_{\bar{c}|\bar{a}}.$$

We will often refer to such a multi–linear function as the *network polynomial.*

Given the network polynomial f, we can answer any query with respect to the belief network. Specifically, let $\mathbf{e}$ be an instantiation of some network variables, and suppose we want to compute the probability of $\mathbf{e}$. We can do this by simply evaluating the polynomial f while setting each evidence indicator λ_x to 1 if x is consistent with $\mathbf{e}$, and to 0 otherwise. For the network in Figure 1, we can compute the probability of evidence $\mathbf{e} = b\bar{c}$ by evaluating its polynomial above under $\lambda_a = 1,\lambda_{\bar{a}} = 1,\lambda_b = 1, \lambda_{\bar{b}} = 0$ and $\lambda_c = 0, \lambda_{\bar{c}} = 1$. This leads to $\theta_a\theta_{b|a}\theta_{\bar{c}|a} + \theta_{\bar{a}}\theta_{b|\bar{a}}\theta_{\bar{c}|\bar{a}}$, which equals the probability of $b, \bar{c}$ in this case. We use $f(\mathbf{e})$ to denote the result of evaluating the polynomial f under evidence $\mathbf{e}$ as given above.

This algebraic representation of belief networks is attractive as it allows us to obtain answers to a large number of probabilistic queries directly from the derivatives of the network polynomial [3]. For example, the posterior marginal $Pr(x|\mathbf{e})$ for a variable $X \not\in \mathbf{E}$ equals $\frac{1}{f(\mathbf{e})}\frac{\partial f(\mathbf{e})}{\partial \lambda_x}$, where $\frac{\partial f(\mathbf{e})}{\partial \lambda_x}$ is the partial derivative of f wrt λ_x evaluated at $\mathbf{e}$. Second, the probability of evidence $\mathbf{e}$ after having retracted the value of some variable X from $\mathbf{e}$, $Pr(\mathbf{e} - X)$, equals $\sum_x \frac{\partial f(\mathbf{e})}{\partial \lambda_x}$. Third, the posterior marginal $Pr(x, \mathbf{u}|\mathbf{e})$ for a variable X and its parents $\mathbf{U}$ equals $\frac{\theta_{x|\mathbf{u}}}{f(\mathbf{e})}\frac{\partial f(\mathbf{e})}{\partial \theta_{x|\mathbf{u}}}$.

The approach presented above is quite transparent semantically as it provides simple closed forms to subtle queries. But it is computationally infeasible since the multi–linear function has an exponential number of terms. One can represent the function compactly in certain cases, however, using an arithmetic circuit; see Figure 2. The (first) partial derivatives of an arithmetic circuit can all be computed simultaneously in time linear in the circuit size [3,12]. The procedure resembles the back–propagation algorithm for neural networks as it evaluates the circuit in a single upward-pass, and then differentiates it through a single downward-pass; see Section 5.

The main computational question is then that of generating the smallest arithmetic circuit that computes the network polynomial. A structure–based approach for this has been given in [3], which is guaranteed to generate a circuit whose size is bounded by $O(n \exp(w))$, where n is the number of nodes in the network and w is its treewidth. A more recent approach, however, which exploits local structure has been presented in [4] and was shown experimentally to generate small arithmetic circuits (a few thousand nodes) for networks

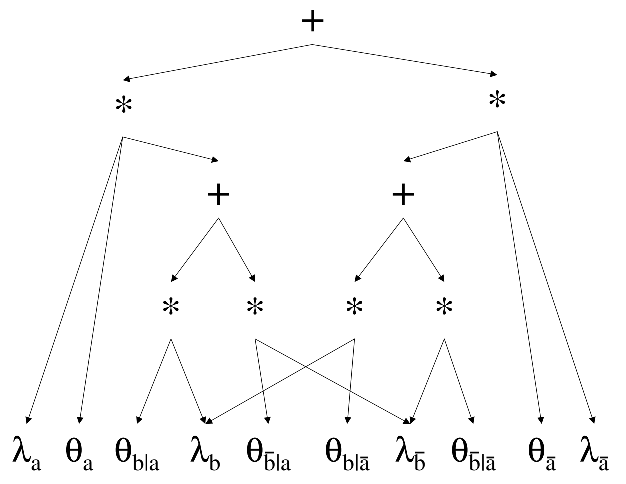


Fig. 2. An arithmetic circuit which computes the function $\lambda_a\lambda_b\theta_a\theta_{b|a}+ \lambda_a\lambda_{\bar{b}}\theta_a\theta_{\bar{b}|a}+ \lambda_{\bar{a}}\lambda_b\theta_{\bar{a}}\theta_{b|\bar{a}}+ \lambda_{\bar{a}}\lambda_{\bar{b}}\theta_{\bar{a}}\theta_{\bar{b}|\bar{a}}$. The circuit is a DAG, where leaf nodes represent function variables and internal nodes represent arithmetic operations.

whose treewidth is up to 60. As we show in the rest of this paper, the process of factoring a belief network into a jointree is yet another method for generating an arithmetic circuit for the network. Specifically, we show that the jointree structure is an implicit representation of such a circuit; that jointree propagation is nothing but an evaluation and differentiation of the embedded circuit; and that the difference between Shenoy–Shafer and Hugin propagation is a difference in the numeric scheme used for circuit differentiation.

3 Jointree Algorithms

We review jointree algorithms in this section, which are very influential algorithms for probabilistic inference in graphical models. Let B be a belief network. A jointree for B is a pair $(\mathbf{T}, \mathbf{L})$, where $\mathbf{T}$ is a tree and $\mathbf{L}$ is a function that assigns labels to nodes in $\mathbf{T}$. A jointree must satisfy three properties: (1) each label $\mathbf{L}(i)$ is a set of variables in the belief network; (2) each network variable X and its parents $\mathbf{U}$ (a family) must appear together in some label $\mathbf{L}(i)$; (3) if a variable appears in the labels of i and j, it must also appear in the label of each node k on the path connecting them.

The label of edge ij in $\mathbf{T}$ is defined as $\mathbf{L}(i) \cap \mathbf{L}(j)$. We will refer to the nodes of a jointree (and sometimes their labels) as *clusters.* We will also refer to the edges of a jointree (and sometimes their labels) as *separators.* Figure 3 depicts a belief network and one of its jointrees.

Jointree algorithms start by constructing a jointree for a given belief network

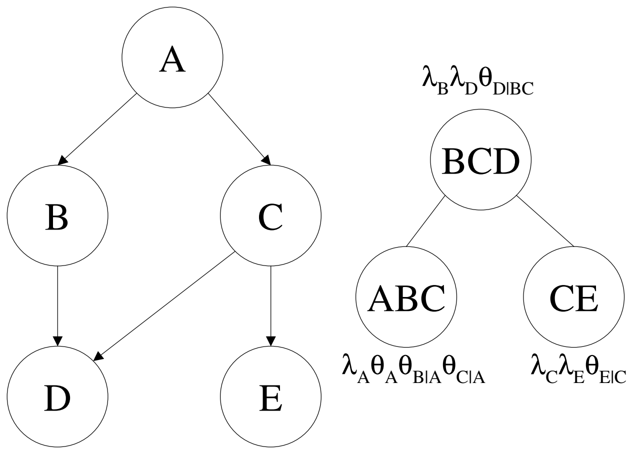

A
B
C
D
E
$\lambda_B\lambda_D\theta_{D|BC}$
BCD
ABC
CE
$\lambda_A\theta_A\theta_{B|A}\theta_{C|A}$
$\lambda_C\lambda_E\theta_{E|C}$

Fig. 3. A belief network and a corresponding jointree.

[14,9,7]. They also associate *tables* (also called *potentials*) with clusters and separators. [3] The *conditional probability table* (CPT or CP Table) of each variable X with parents $\mathbf{U}$, denoted $\theta_{X|\mathbf{U}}$, is assigned to a cluster that contains X and $\mathbf{U}$. In addition, an *evidence table* over variable X, denoted λ_X, is assigned to a cluster that contains X. Figure 3 depicts the assignments of evidence and CP tables to clusters. Evidence $\mathbf{e}$ is entered into a jointree by initializing evidence tables as follows: we set $\lambda_X(x)$ to 1 if x is consistent with evidence $\mathbf{e}$, and we set $\lambda_X(x)$ to 0 otherwise.

Given some evidence $\mathbf{e}$, a jointree algorithm propagates messages between clusters. After passing two messages per edge in the jointree, one can compute the marginals $Pr(\mathbf{C}, \mathbf{e})$ for every cluster $\mathbf{C}$. There are two main methods for propagating messages in a jointree, known as the Shenoy–Shafer architecture [14] and the Hugin architecture [9].

Shenoy–Shafer propagation proceeds as follows [14]. First, evidence $\mathbf{e}$ is entered into the jointree. A cluster is then selected as the root and message propagation proceeds in two phases, inward and outward. In the inward phase, messages are passed toward the root. In the outward phase, messages are passed away from the root. Cluster i sends a message to cluster j only when it has received messages from all its other neighbors k. A message from cluster i to cluster j is a table M_{ij} defined as follows:

$$M_{ij} = \sum_{\mathbf{C}\setminus\mathbf{S}} \phi_i \prod_{k\neq j} M_{ki},$$

[3] A table is an array which is indexed by variable instantiations. Specifically, a table ϕ over variables $\mathbf{X}$ is indexed by the instantiations $\mathbf{x}$ of $\mathbf{X}$. Its entries $\phi(\mathbf{x})$ are in $[0, 1]$. We assume familiarity with table operations, such as multiplication and marginalization [7].

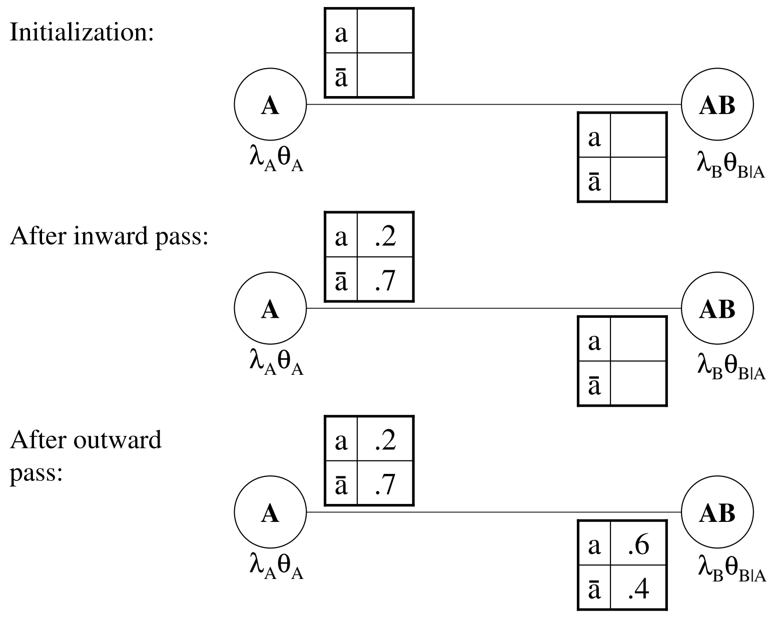


Fig. 4. Shenoy–Shafer propagation illustrated on a simple jointree under evidence b. The jointree is for network $A \rightarrow B$, where $\theta_a = .6$, $\theta_{b|a} = .2$ and $\theta_{b|\bar{a}} = .7$.

where $\mathbf{C}$ are the variables of cluster i, $\mathbf{S}$ are the variables of separator ij, and ϕ_i is the multiplication of all evidence and CP tables assigned to cluster i. Once message propagation is finished, we have $Pr(\mathbf{C}, \mathbf{e}) = \phi_i \prod_k M_{ki}$, where $\mathbf{C}$ are the variables of cluster i. Figure 4 illustrates Shenoy–Shafer propagation on a simple example.

The space requirements for the Shenoy–Shafer architecture are those needed to store the messages. For each separator ij with variables $\mathbf{S}$, we need two tables over variables $\mathbf{S}$. One table stores the message from cluster i to cluster j, and the other stores the message from j to i.

Hugin propagation proceeds similarly to Shenoy–Shafer by entering evidence; selecting a cluster as root; and propagating messages in two phases, inward and outward [9]. The Hugin method, however, differs in some major ways. It maintains a table Φ_{ij} with each separator, whose entries are initialized to 1s. It also maintains a table Φ_i with each cluster i, initialized to the multiplication of all CPTs and evidence tables assigned to cluster i; see Figure 5. Cluster i passes a message to neighboring cluster j only when i has received messages from all its other neighbors k. When cluster i is ready to send a message to cluster j, it does the following. First, it saves the table of separator Φ_{ij} into Φ_{ij}^{old}. Second, it computes a new separator table $\Phi_{ij} = \sum_{\mathbf{C} \setminus \mathbf{S}} \Phi_i$, where $\mathbf{C}$ are the variables of cluster i and $\mathbf{S}$ are the variables of separator ij. Third, it computes a message to cluster j: $M_{ij} = \frac{\Phi_{ij}}{\Phi_{ij}^{old}}$. Finally, it multiplies the computed message into the table of cluster j: $\Phi_j = \Phi_j M_{ij}$. After the inward and outward–passes of Hugin

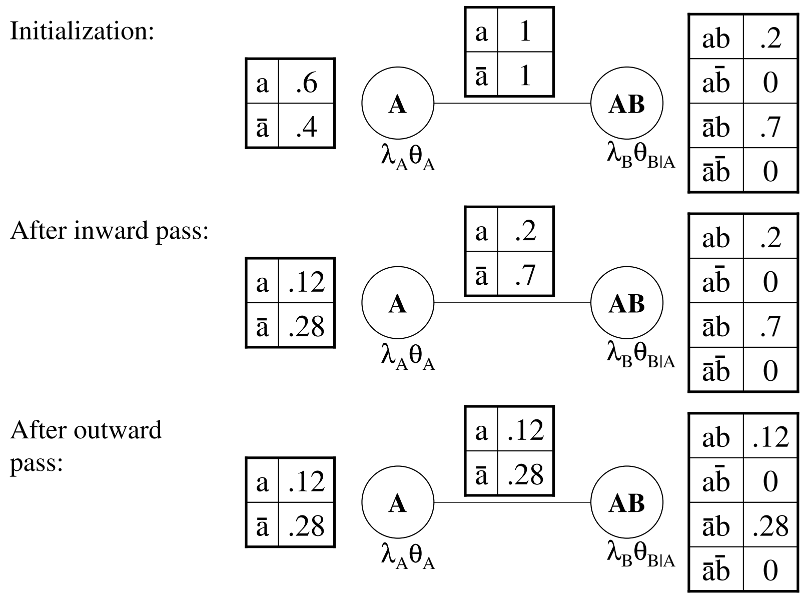


Fig. 5. Hugin propagation illustrated on a simple jointree under evidence b. The jointree is for network $A \rightarrow B$, where $\theta_a = .6$, $\theta_{b|a} = .2$ and $\theta_{b|\bar{a}} = .7$.

propagation are completed, we have: $Pr(\mathbf{C}, \mathbf{e}) = \Phi_i$, where $\mathbf{C}$ are the variables of cluster i. Figure 5 illustrates Hugin propagation on a simple example.

Therefore, the space requirements for the Hugin architecture are those needed to store cluster and separator tables: one table for each cluster and one table for each separator. Note that the Hugin architecture does not save the messages exchanged between clusters.

4 Jointrees as arithmetic circuits

We now show that every jointree (together with a root cluster and a particular assignment of evidence and CP tables to clusters) corresponds precisely to an arithmetic circuit that computes the network polynomial. We also show that the inward–pass of the Shenoy–Shafer architecture evaluates this circuit, while the outward–pass differentiates it. We show a similar result for the Hugin architecture. Interestingly enough, the difference between the two architectures can be viewed as a difference in the numeric scheme used to implement circuit differentiation, which is explained in Section 5.

We now define the arithmetic circuit embedded in a jointree. Given a root cluster, one can direct the jointree by having arrows point away from the root, which also defines a parent/child relationship between clusters and separators.

Definition 1 *Given a root cluster, a particular assignment of evidence and CP tables to clusters, the arithmetic circuit embedded in a jointree is defined as follows:*

Nodes: The circuit includes: an output addition node f; an addition node $\mathbf{s}$ for each instantiation of a separator $\mathbf{S}$; a multiplication node $\mathbf{c}$ for each instantiation of a cluster $\mathbf{C}$; an input node λ_x for each instantiation x of variable X; an input node $\theta_{x|\mathbf{u}}$ for each instantiation $x\mathbf{u}$ of family $X\mathbf{U}$.

Edges: The children of the output node f are the multiplication nodes generated by the root cluster; the children of an addition node $\mathbf{s}$ are all compatible nodes generated by the child cluster; the children of a multiplication node $\mathbf{c}$ are all compatible nodes generated by child separators; in addition to all compatible inputs nodes corresponding to cluster $\mathbf{C}$.

Hence, separators contribute addition nodes and clusters contribute multiplication nodes. Moreover, the structure of the jointree dictates how these nodes are connected into a circuit. The arithmetic circuit in Figure 2 is embedded in the jointree $A - AB$, with cluster A as the root, and with tables λ_A, θ_A assigned to cluster A and tables λ_B and $\theta_{B|A}$ assigned to cluster B. There are three addition nodes in this circuit, two of which correspond to the instantiations of separator A. There are also six multiplication nodes, the top two correspond to cluster A and the bottom four correspond to cluster AB. Note that the arithmetic circuit embedded in a jointree has a very specific structure: it alternates between addition and multiplication nodes; its output is always an addition node; and every multiplication node has a unique parent.

Obviously, the inputs of the arithmetic circuit embedded in a jointree are in one–to–one correspondence with variables in the network polynomial. The following theorem says that the circuit and network polynomial represent the same function.

Theorem 1 *The arithmetic circuit embedded in a jointree computes the network polynomial.*

Therefore, by constructing a jointree one is generating a compact representation of the network polynomial in terms of an arithmetic circuit, where Definition 1 describes precisely how to obtain such a circuit from the constructed jointree. Note that the number of addition and multiplication nodes in the circuit equals the number of cluster and separator entries plus 1.

4.1 Differential semantics

We are now ready to state our basic results on the differential semantics of jointree propagation, but we need some notational conventions first. In the following three theorems: f denotes the circuit embedded in a jointree or its (unique) output node; $\mathbf{s}$ denotes a separator instantiation or the addition node generated by that instantiation; and $\mathbf{c}$ denotes a cluster instantiation or the multiplication node generated by that instantiation. Moreover, the value that a circuit node v takes under evidence $\mathbf{e}$ is denoted $v(\mathbf{e})$. Recall that a circuit (or network polynomial) is evaluated under evidence $\mathbf{e}$ by setting each input λ_x to 1 if x is consistent with $\mathbf{e}$; and to 0 otherwise. Finally, recall that $\partial f / \partial v$ represents the derivative of the circuit output with respect to node v.

Our first result relates to Shenoy–Shafer propagation.

Theorem 2 *The messages produced using Shenoy–Shafer propagation on an arbitrary jointree under evidence* $\mathbf{e}$ *have the following semantics.*

For each inward message M_{ij}*, we have* $M_{ij}(\mathbf{s}) = \mathbf{s}(\mathbf{e})$.
For each outward message M_{ji}*, we have* $M_{ji}(\mathbf{s}) = \frac{\partial f(\mathbf{e})}{\partial \mathbf{s}}$.

That is, if we interpret separator instantiations as addition nodes in a circuit as given by Definition 1, we get that a message directed towards the jointree root contains the values of these addition nodes, while a message directed outward from the root contains the partial derivatives of the circuit output with respect to these addition nodes.

Shenoy–Shafer propagation does not compute derivatives with respect to input nodes λ_x and $\theta_{x|\mathbf{u}}$, but these can be obtained using local computations as follows.

Theorem 3 *Suppose that evidence table* λ_X *is assigned to cluster* i *which has variables* $\mathbf{C}$*. Then:*

$$\frac{\partial f(\mathbf{e})}{\partial \lambda_x} = \left[\sum_{\mathbf{C} \setminus X} \prod_j M_{ji} \prod_{\psi \neq \lambda_X} \psi \right](x), \tag{1}$$

where ψ *ranges over all evidence and CP tables assigned to cluster* i*. Suppose now that CPT* $\theta_{X|\mathbf{U}}$ *is assigned to cluster* i *which has variables* $\mathbf{C}$*. Then:*

$$\frac{\partial f(\mathbf{e})}{\partial \theta_{x|\mathbf{u}}} = \left[\sum_{\mathbf{C} \setminus X\mathbf{U}} \prod_j M_{ji} \prod_{\psi \neq \theta_{X|\mathbf{U}}} \psi \right](x\mathbf{u}), \tag{2}$$

where ψ ranges over all evidence and CP tables assigned to cluster i.

Therefore, even though Shenoy–Shafer propagation does not fully differentiate the embedded arithmetic circuit, the differentiation process can be completed through local computations after propagation has finished. The extra derivatives computed in this process are quite valuable as we discuss later.

We now present a similar, but more extensive results on Hugin propagation.

Theorem 4 *Cluster tables, separator tables and messages produced using Hugin propagation under evidence $\mathbf{e}$ have the following semantics:*

For table Φ_i of cluster i with variables $\mathbf{C}$: $\Phi_i(\mathbf{c}) = \mathbf{c}(\mathbf{e})\dfrac{\partial f(\mathbf{e})}{\partial \mathbf{c}}$.
For table Φ_{ij} of separator ij with variables $\mathbf{S}$: $\Phi_{ij}(\mathbf{s}) = \mathbf{s}(\mathbf{e})\dfrac{\partial f(\mathbf{e})}{\partial \mathbf{s}}$.
For each inward message M_{ij}, we have $M_{ij}(\mathbf{s}) = \mathbf{s}(\mathbf{e})$.
For each outward message M_{ji}, we have $M_{ji}(\mathbf{s}) = \frac{\partial f(\mathbf{e})}{\partial \mathbf{s}}$ if $\mathbf{s}(\mathbf{e}) \neq 0$.

Again, Hugin propagation does not compute derivatives with respect to input nodes λ_x and $\theta_{x|\mathbf{u}}$. Even for addition and multiplication nodes, it only retains derivatives multiplied by values.[4] Hence, if we want to recover the derivative with respect to, say, multiplication node $\mathbf{c}$, we must know the value of this node and it must be different than zero. In such a case, we have $\partial f(\mathbf{e})/\partial \mathbf{c} = \Phi_i(\mathbf{c})/\mathbf{c}(\mathbf{e})$, where Φ_i is the table associated with the cluster i that generates node $\mathbf{c}$. One can also compute the quantity $v\ \partial f/\partial v$ for input nodes using equations similar to those in Theorem 3. But such quantities will be useful for obtaining derivatives only if the values of such input nodes are not zero. Hence, Shenoy–Shafer propagation is more informative than Hugin propagation as far as computing derivatives is concerned.

4.2 Applications of derivatives

Partial derivatives with respect to evidence indicators λ_x and network parameters $\theta_{x|\mathbf{u}}$ have many applications, and our ability to compute them using standard jointree propagation has been unveiled by Theorem 3. We discuss these applications next.

[4] Hugin propagation computes derivatives with respect to addition nodes at some point (outward messages) but does not save them.

Fast retraction & evidence flipping

Suppose jointree propagation has been performed using evidence $\mathbf{e}$, which gives us access directly to the probability of $\mathbf{e}$. Suppose now we are interested in the probability of a different evidence $\mathbf{e}'$, which results from changing the value of some variable X in $\mathbf{e}$ to a new value x. The probability of $\mathbf{e}'$ in this case is equal to $\frac{\partial f(\mathbf{e})}{\partial \lambda_x}$ [3], which can be obtained as given by Equation 1. The ability to perform this computation efficiently is crucial for algorithms that try to approximate *maximum a posteriori hypothesis* (MAP) using local search [10,11]. Another application of this derivative is in computing the probability of evidence $\mathbf{e}'$, which results from retracting the value of some variable X from $\mathbf{e}$: $Pr(\mathbf{e}') = \sum_x \frac{\partial f(\mathbf{e})}{\partial \lambda_x}$. This computation is key to analyzing evidence conflict, as it allows us to determine the extent to which one piece of evidence is contradicted by the remaining pieces.

Sensitivity analysis & parameter learning

The derivative $\frac{\partial Pr(\mathbf{e})}{\partial \theta_{x|\mathbf{u}}}$ is essential for sensitivity analysis—it is the basis for an efficient approach that identifies minimal network parameters changes that are necessary to satisfy constraints on probabilistic queries [1]. This derivative is also crucial for gradient ascent approaches for learning network parameters as it is required to compute the gradient used for deciding moves in the search space [13]. This derivative equals $\frac{\partial f(\mathbf{e})}{\partial \theta_{x|\mathbf{u}}}$, and can be obtained as given by Equation 2. The only other method we are aware of to compute this derivative (beyond the one in [3]) is the one using the identity $\partial Pr(\mathbf{e})/\partial \theta_{x|\mathbf{u}} = Pr(x, \mathbf{u}, \mathbf{e})/\theta_{x|\mathbf{u}}$, which requires $\theta_{x|\mathbf{u}} \neq 0$ [13]. Hence, our results seem to suggest the first general approach for computing this derivative using standard jointree propagation.

Bounding rounding errors

Jointree propagation gives exact results only when infinite precision arithmetic is used. In practice, however, finite precision floating–point arithmetic is typically used. The differential semantics of jointree propagation allows us to bound the rounding error in the computed probability of evidence, under a particular model of error propagation. Specifically, note that during the inward–pass of Hugin propagation, the value of an entry in separator table Φ_{ij} is simply the addition of compatible entries in the cluster table Φ_i. Moreover, the value of an entry in a cluster table Φ_i is simply the multiplication of its initial value with all corresponding entries in neighboring separators. Let δ be the local rounding error generated when computing the value of a cluster or separator. It is reasonable to assume that $|\delta| \leq \epsilon|v|$, where v is the cluster/separator entry we would have computed using (local) infinite–precision computation, and ϵ is a constant representing the machine–specific relative error occurring in the floating–point representation of a real number [8]. Let us finally assume that the probability of evidence is computed by summing the entries of the root cluster (after the inward–pass of Hugin propagation has finished). We can then bound the rounding error in the computed probability of evidence by:

$$\epsilon \left(\sum_{\mathbf{c}} \Phi_r(\mathbf{c}) + \sum_{i,\mathbf{c}} \Phi_i(\mathbf{c}) + \sum_{ij,\mathbf{s}} \Phi_{ij}(\mathbf{s}) \right), \tag{3}$$

where r is the root cluster. Interestingly enough, this bound can be computed easily during the outward–pass of Hugin propagation. The bound follows immediately assuming a model of error propagation where the total error in computing quantity f is $\sum_v \delta v \partial f / \partial v$, where v ranges over all intermediate computations and δv is the local rounding error generated when performing the intermediate computation v [8].

A similar bound for the rounding error in Shenoy–Shafer propagation can also be derived under similar assumptions.

5 Evaluating and differentiating arithmetic circuits

Our goal in this section is to discuss the evaluation and differentiation of arithmetic circuits, given a particular circuit input. Evaluation is straightforward, but differentiation can be a bit more involved. We discuss two main results relating to differentiation. First, that the partial derivative of the circuit output with respect to each and every circuit node can all be computed in time linear in the circuit size. Second, we discuss three different numeric schemes for differentiating a circuit that vary mainly in their space requirements. One of these methods corresponds to the outward phase of Shenoy–Shafer propagation, and another corresponds to the outward phase of Hugin propagation. Therefore, given our interpretation of jointree propagation as circuit evaluation and differentiation, we now understand the difference between Shenoy–Shafer and Hugin as a difference in how differentiation is performed.

Basic method for circuit differentiation

In what follows, we will not distinguish between an arithmetic circuit and its unique output node. Let f be an arithmetic circuit and let v be one of its nodes. We are interested in the partial derivative of f with respect to node

v, $\partial f/\partial v$. We will first discuss a general, basic method for computing such derivatives, which requires that we store two numbers with each circuit node v:

- $vr(v)$: stores the value of node v;
- $dr(v)$: stores the derivative of f with respect to v.

We will then discuss two other methods that require less space, but are valid only for a specific class of arithmetic circuits (including those generated by jointrees).

The key observation is to view the circuit f as a function of each and every circuit node v. If v is the root node (circuit output), then $\frac{\partial f}{\partial v} = 1$. If v is not the root node, and has parents p, then by the chain rule:

$$\frac{\partial f}{\partial v} = \sum_p \frac{\partial f}{\partial p}\frac{\partial p}{\partial v}.$$

Suppose now that v' are the other children of parent p. If parent p is a multiplication node,

$$\frac{\partial p}{\partial v} = \frac{\partial(v \prod v')}{\partial v} = \prod v'.$$

Similarly, if parent p is an addition node,

$$\frac{\partial p}{\partial v} = \frac{\partial(v + \sum v')}{\partial v} = 1.$$

With these equations, we can recursively compute the partial derivatives of f with respect to any node v in the circuit in time linear in the size of the circuit [3]. The procedure is described below in terms of two passes. An upward–pass which evaluates the circuit by setting the values of $vr(v)$ registers, and a downward–pass which sets the values of $dr(v)$ registers ($dr(v)$ is initialized to zero except that $dr(v) = 1$ for root v). From here on, when we say an upward–pass of the circuit, we will mean a traversal of the circuit where the children of a node are visited before the node itself is visited. Similarly, in a downward–pass, the parents of a node will be visited first.

- Upward–pass: At node v, compute the value of v and store it in $vr(v)$.
- Downward–pass: At node v and for each parent p, increment $dr(v)$
  - by $dr(p)$ if p is an addition node;
  - by $dr(p) \prod vr(v')$ if p is a multiplication node, where v' are the other children of p.

Therefore, a single upward–pass through the circuit will evaluate it, and a single downward–pass will compute all its derivatives. It should be clear that the upward–pass takes time linear in the circuit size, where size is defined as the number of circuit edges. The downward–pass, however, is only linear in case each multiplication node has a bounded number of children, which would ensure that the expression $dr(p) \prod vr(v')$ takes bounded time to evaluate. We can always convert an arithmetic circuit into one where each multiplication node has two children, while only increasing its size by a linear factor. [5] But a more sophisticated approach is described in [2], which attains linearity without increasing the circuit size.

The basic differentiation method described above uses two registers per circuit node and is the one used in [3]. The circuits generated from jointrees have specific properties, however, which allow us to do better in terms of space usage. Specifically, these circuits alternate between addition and multiplication nodes; the output node is always an addition node; and each multiplication node has a single parent. Given these properties, one can employ a differentiation scheme that uses less space than is suggested above. We discuss two such methods in the following sections:

- *Method A:* Requires two registers for each addition node, and no registers for multiplication nodes.
- *Method B:* Requires only one register for each addition and multiplication node.

Method A for circuit differentiation

This method uses two registers $vr(v)$ and $dr(v)$ for every addition node v. Registers are initialized to zero except for the root v where $dr(v) = 1$. Only multiplication nodes v are visited during each pass, where node v must have a unique parent p:

- *Upward-pass:* At multiplication node v, compute the value of v and accumulate the result into $vr(p)$.
- *Downward-pass:* At multiplication node v, for each addition child c, compute $dr(p) \prod vr(c')$ and accumulate the result into $dr(c)$, where c' ranges over other children of v.

Figure 6 depicts the contents of vr and dr registers after each pass of Method A on a simple circuit. Note that similar to the basic method, the downward–pass

[5] Basically, if v is a multiplication node with n children, we can replace v by a sequence of multiplication nodes each with only two children. This increases the number of nodes in the circuit by $n - 2$ and the number of edges by $n - 2$.

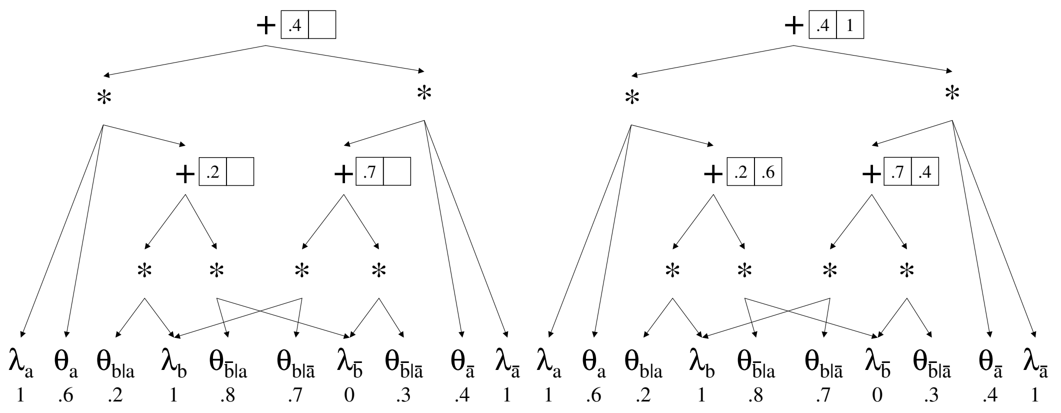


Fig. 6. On the left, an arithmetic circuit after the upward–pass of Method A. On the right, the same circuit after the downward–pass of Method A. vr registers are shown on the left, and dr registers on the right.

of Method A is not necessarily linear if the number of children per multiplication node is not bounded. The correspondence of this method to Shenoy–Shafer propagation is given next.

Theorem 5 *Let f be the embedded circuit in jointree J and let* $\mathbf{s}$ *be an addition node in f which corresponds to instantiation* $\mathbf{s}$ *of separator ij. After Shenoy–Shafer propagation and Method A passes are finished, we have $M_{ij}(\mathbf{s}) = vr(\mathbf{s})$ and $M_{ji}(\mathbf{s}) = dr(\mathbf{s})$, where cluster j is closer to the root than cluster i.*

Method A can afford to take much less space than the basic method due to the following property. If v is a multiplication node, then v must have a single parent p, which is an addition node. Hence, $\partial f/\partial v = \partial f/\partial p$, and the method avoids using the register $dr(v)$ as it will have the same value as $dr(p)$. It also never stores the value of v explicitly: once that value is computed, it is immediately accumulated at the parent p in register $vr(p)$.

Finally, note that this method does not compute derivatives with respect to input nodes v, but such derivatives can be computed easily if need be (see Theorem 3).

Method B for circuit differentiation

We now discuss another differentiation scheme with reduced space requirements that works differently than Method A. This method is used by the outward–pass of Hugin propagation and uses only one register $r(v)$ for each addition and multiplication node. Specifically, Method B evaluates the circuit in a classical way, storing the computed value of each node v in register $r(v)$.

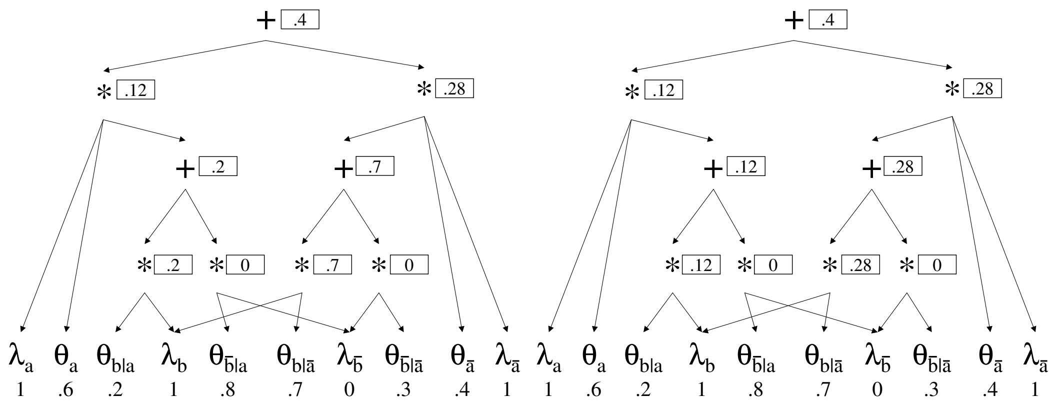


Fig. 7. On the left, an arithmetic circuit after the upward–pass of Method B. On the right, the same circuit after the downward–pass of Method B.

But it overrides these values in the downward–pass, where the value of each node is replaced by $v\ \partial f/\partial v$. Only addition nodes (except the root) are visited in the downward–pass:[6]

- *Upward–pass:* At node v, compute the value of v and store it in $r(v)$.
- *Downward–pass:* At addition node $v \neq f$ with parents p and multiplication children c:
  · save $r(v)$ into old;
  · reset $r(v)$ to $\sum_p r(p)$;
  · multiply $r(v)/old$ into $r(c)$ for each child c.

Figure 7 depicts the contents of r registers after each pass of Method B on a simple arithmetic circuit. Method B takes time linear in the circuit size. Its correspondence to Hugin propagation is given next.

Theorem 6 *Let f be the embedded circuit in jointree J, $\mathbf{c}$ be a multiplication node in f corresponding to instantiation $\mathbf{c}$ of cluster i, and $\mathbf{s}$ be an addition node in f corresponding to instantiation $\mathbf{s}$ of separator ij. After Hugin propagation and Method B passes are finished, we have $\Phi_i(\mathbf{c}) = r(\mathbf{c})$ and $\Phi_{ij}(\mathbf{s}) = r(\mathbf{s})$.*

The main insight behind Method B is as follows. If v is a node with a multiplication parent p, then $p = v\ \partial p/\partial v$. Now, if all parents p of v are multiplication nodes, then $\partial f/\partial v = \sum_p (\partial f/\partial p)(\partial p/\partial v)$ by the chain rule. Multiplying both sides by v, we get the important identity:

$$v\ \partial f/\partial v = \sum_p (\partial f/\partial p)\ v\ (\partial p/\partial v) = \sum_p p\ \partial f/\partial p.$$

[6] Note that 0/0 is defined to be 0.

Now when visiting addition node v, assume by induction that the register $r(p)$ of its parent p contains $p\ \partial f/\partial p$. The register of v is then guaranteed to contain $v\ \partial f/\partial v$ when it is reset as given above. Moreover, if c is a multiplication child of v, then v must be the only parent of c, leading to $\partial f/\partial c = \partial f/\partial v = r(v)/r^{old}(v)$. Therefore, after node v is processed, the register $r(c)$ of each of its children c will contain $c\ \partial f/\partial c$. And when the downward–pass is completely over, the register $r(v)$ of every node will contain $v\ \partial f/\partial v$.

This method takes time which is linear in the circuit size. It does not compute the value $v\ \partial f/\partial v$ for input nodes, but that can be computed easily. The problem, however, is that this quantity is not useful for obtaining $\partial f/\partial v$ unless $v \neq 0$. Therefore, this method is limited compared to Method A as it allows us to compute derivatives only when nodes have non-zero values.

6 A new perspective on factoring graphical models

We have shown in this paper that each jointree can be viewed as an implicit representation of an arithmetic circuit that computes the network polynomial. Moreover, we have shown that jointree propagation corresponds to an evaluation and differentiation of the circuit, where the difference between Shenoy–Shafer and Hugin propagation amounts to a difference between the way they carry out the differentiation task. These results have been useful in unifying the circuit approach presented in [3] with jointree approaches, and in uncovering more properties of jointree propagation.

Another outcome of these results relates to the level at which it is useful to phrase the problem of factoring graphical probabilistic models. Specifically, the perspective we are promoting here is that probability distributions defined by graphical models should be viewed as multi–linear functions, and the construction of jointrees should be viewed as a process of constructing arithmetic circuits that compute these functions. That is, the fundamental object being factored is a multi–linear function, and the fundamental result of the factorization is an arithmetic circuit. A graphical model is a useful abstraction of the multi–linear function, and a jointree is a useful structure for embedding the arithmetic circuit.

This view of factoring is useful since it allows us to cast the factoring problem in more refined terms, which puts us in a better position to exploit the local structure of graphical models in the factorization process. Note that the topology of a graphical model defines the form of the multi–linear function, while the model's local structure (as exhibited in its quantification) constrains the values of variables appearing in the function. One can factor a multi–linear function without knowledge of such constraints, but the resulting factorizations will not be optimal. For a dramatic example, consider a fully connected network with variables $X_1, \ldots, X_n$, where all parameters are equal $(\frac{1}{2})$. Any jointree for the network will have a cluster of size n, leading to $O(\exp(n))$ complexity. There is, however, a circuit of $O(n)$ size here, since the network polynomial can be easily factored as: $f = (\frac{1}{2})^n \prod_{i=1}^{n}(\lambda_{x_i} + \lambda_{\bar{x}_i})$.

Hence, in the presence of local structure, it appears more promising to factor the graphical model into the more refined arithmetic circuit since not every arithmetic circuit can be embedded in a jointree. This promise is made apparent by the results in [4], which we sketch next. First, the multi–linear function of a belief network is "encoded" using a propositional theory, which is expressive enough to capture the form of the multi–linear function in addition to constraints on its variables. The theory is then compiled into a special logical form, known as decomposable negation normal form. An arithmetic circuit is finally extracted from that form. The method was able to generate relatively small arithmetic circuits for a significant suite of real-world belief networks with treewidths up to 60. [7]

It worth mentioning here that the above perspective is in harmony with recent approaches that represent probabilistic models using algebraic decision diagrams (ADDs), citing the promise of ADDs in exploiting local structure [6]. ADDs and related representations, such as edge–valued decision diagrams, are known to be compact representations of multi–linear functions. Moreover, each of these representations can be expanded in linear time into an arithmetic circuit that satisfies some strong properties [5]. Hence, such representations are special cases of arithmetic circuits, which leads to practical implications that are explored in [5].

We finally note that the relationship between multi–linear functions (polynomials in general) and arithmetic circuits is a classical subject of *algebraic complexity theory* [15]. In this field of complexity, computational problems are expressed as polynomials, and a central question is that of determining the size of the smallest arithmetic circuit that computes a given polynomial, leading to the notion of *circuit complexity.* Using this notion, it is then meaningful to talk about the circuit complexity of a graphical model: the size of the smallest arithmetic circuit that computes the multi–linear function induced by the model.

[7] Refinements on jointree methods, such as zero compression, take advantage of local structure such as determinism. They can be interpreted as identifying circuit nodes that are stuck to zero and eliminating them. One problem with these methods, however, is that they perform full propagation with respect to the complete jointree before they are able to compress it. Hence, they cannot handle networks like those in [4].

Acknowledgment

This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

[1] Hei Chan and Adnan Darwiche. When do numbers really matter? In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 65–74, San Francisco, California, 2001. Morgan Kaufmann Publishers, Inc.

[2] Adnan Darwiche. A differential approach to inference in Bayesian networks. Technical Report D–108, Computer Science Department, UCLA, Los Angeles, Ca 90095, 1999. To appear in the Journal of ACM.

[3] Adnan Darwiche. A differential approach to inference in Bayesian networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 123–132, 2000. To appear in Journal of ACM.

[4] Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.

[5] Adnan Darwiche. On the factorization of multi-linear functions. Technical Report D–128, Computer Science Department, UCLA, Los Angeles, Ca 90095, 2002.

[6] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288, 1999.

[7] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[8] Masao Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding error. *Japan J. Appl. Math.*, 1:223–252, 1984.

[9] F. V. Jensen, S.L. Lauritzen, and K.G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[10] James Park. MAP complexity results and approximation methods. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 388–396, San Francisco, California, 2002. Morgan Kaufmann Publishers, Inc.

[11] James Park and Adnan Darwiche. Approximating MAP using local search. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 403–410, San Francisco, California, 2001. Morgan Kaufmann Publishers, Inc.

[12] Graz Rote. Path problems in graphs. *Computing Suppl.*, 7:155–189, 1990.

[13] S. Russell, J. Binder, D. Koller, and K. Kanazawa. Local learning in probabilistic networks with hidden variables. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 1146–1152, 1995.

[14] Prakash P. Shenoy and Glenn Shafer. Propagating belief functions with local computations. *IEEE Expert*, 1(3):43–52, 1986.

[15] J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comp. Sci.*, 3:317–347, 1988.

A Proofs of Theorems

Proof of Theorem 1

Consider the following method for computing the probability of evidence in a jointree. First, choose a root cluster i and poll messages towards the root. Second, compute the probability of evidence as $\sum_{\mathbf{c}} \prod_j M_{ji}$, where $\mathbf{C}$ are the variables of cluster i. It is not hard to realize that the circuit embedded in a jointree represents a trace of the previous computation. $\square$

Proof of Theorem 2

Follows immediately from the correspondence of Shenoy–Shafer propagation to Method A for circuit differentiation as given by Theorem 5. $\square$

Proof of Theorem 3

Suppose that λ_X is assigned to cluster i in the given jointree J. Let us augment the jointree by an additional cluster k which contains only variable X, leading to jointree J'. Make cluster k a neighbor of cluster i in jointree J' and assign evidence table λ_X to cluster k (instead of cluster i). Note that the separator between clusters i and k will have a single variable X. Moreover, in the embedded circuit of jointree J', λ_x will have a single multiplication node m as a parent, and m will have a single addition node n as a parent, which corresponds to the instantiation x of separator X. Hence, $\partial f/\partial\lambda_x = \partial f/\partial m = \partial f/\partial n$. Moreover, given Theorem 2, we have $\partial f/\partial n = M_{ik}(x)$ with respect to jointree J', which is equal to $[\sum_{\mathbf{C}\setminus X} \prod_j M_{ji} \prod_{\psi \neq \lambda_X} \psi](x)$ in the original jointree J as given by Equation 1. The proof for Equation 2 is similar. $\square$

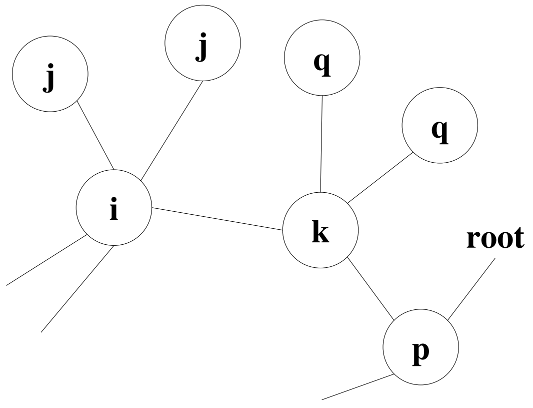


Fig. A.1. Clusters in a jointree.

Proof of Theorem 4

Follows immediately from the correspondence of Hugin propagation to Method B for circuit differentiation as given by Theorem 6. □

Proof of Theorem 5

First, there is a one-to-one correspondence between addition nodes (except the root) in the embedded circuit f and instantiations of separators in jointree J. The same is true for multiplication nodes and instantiations of clusters.

Part 1

Let i be a cluster, k be its neighboring cluster closest to the root, and j be other neighboring clusters; see Figure A.1. Let $\mathbf{s}_{ki}$, $\mathbf{s}_{ji}$ and $\mathbf{c}_i$ be instantiations of the corresponding separators and clusters. Also, let τ_i be all evidence indicators and network parameters assigned to cluster i. In the circuit f, the children of addition node $\mathbf{s}_{ki}$ will be all compatible multiplication nodes $\mathbf{c}_i$. Moreover, the children of multiplication node $\mathbf{c}_i$ will be all compatible addition nodes $\mathbf{s}_{ji}$, in addition to all compatible evidence indicators and network parameters τ_i. According to Shenoy–Shafer propagation, we have

$$M_{ik}(\mathbf{s}_{ik}) = \sum_{\mathbf{c}_i \sim \mathbf{s}_{ik}} \prod_{j,\, \mathbf{s}_{ji} \sim \mathbf{c}_i} M_{ji}(\mathbf{s}_{ji}) \prod_{\tau_i \sim \mathbf{c}_i} \tau_i,$$

where $\sim$ denotes the compatibility relation among instantiations. Assume by induction that each register $vr(\mathbf{s}_{ji})$ will contain $M_{ji}(\mathbf{s}_{ji})$ after the message M_{ji} has been computed. It then follows that register $vr(\mathbf{s}_{ik})$ will contain $M_{ik}(\mathbf{s}_{ik})$

after the message M_{ik} has been computed. The base case for this induction is when cluster i has a single neighbor k, where Shenoy–Shafer propagation gives:

$$M_{ik}(\mathbf{s}_{ik}) = \sum_{\mathbf{c}_i \sim \mathbf{s}_{ik}} \prod_{\tau_i \sim \mathbf{c}_i} \tau_i,$$

which is immediately equal to $vr(\mathbf{s}_{ik})$.

Part 2

Now let p be the neighbor of cluster k closest to the root, and let q be other neighbors of k where $p \neq i$ and $q \neq i$; see Figure A.1. The parents of addition node $\mathbf{s}_{ki}$ will be all compatible multiplication nodes $\mathbf{c}_k$. Moreover, each multiplication node $\mathbf{c}_k$ will have a single parent, which is the compatible addition nodes $\mathbf{s}_{pk}$. The other children of $\mathbf{c}_k$, beyond the compatible $\mathbf{s}_{ki}$, will be all compatible addition nodes $\mathbf{s}_{qk}$ and compatible inputs τ_k. Now, according to Shenoy–Shafer propagation, we have

$$M_{ki}(\mathbf{s}_{ki}) = \sum_{\mathbf{c}_k \sim \mathbf{s}_{ki}, \mathbf{s}_{pk} \sim \mathbf{c}_k} M_{pk}(\mathbf{s}_{pk}) \prod_{q,\, \mathbf{s}_{qk} \sim \mathbf{c}_k} M_{qk}(\mathbf{s}_{qk}) \prod_{\tau_k \sim \mathbf{c}_k} \tau_k.$$

According to Method A,

$$dr(\mathbf{s}_{ki}) = \sum_{\mathbf{c}_k \sim \mathbf{s}_{ki}, \mathbf{s}_{pk} \sim \mathbf{c}_k} dr(\mathbf{s}_{pk}) \prod_{q,\, \mathbf{s}_{qk} \sim \mathbf{c}_k} vr(\mathbf{s}_{qk}) \prod_{\tau_k \sim \mathbf{c}_k} \tau_k.$$

We have proven in Part 1 that $vr(\mathbf{s}_{qk}) = M_{qk}(\mathbf{s}_{qk})$. If we also assume by induction that register $dr(\mathbf{s}_{pk})$ contains $M_{pk}(\mathbf{s}_{pk})$ after message M_{pk} has been computed, it follows then that register $dr(\mathbf{s}_{ki})$ must contain $M_{ki}(\mathbf{s}_{ki})$ after message M_{ki} has been computed. The base case for this induction is when cluster k is the root. We no longer have the special neighbor q in this case, and Shenoy–Shafer gives:

$$M_{ki}(\mathbf{s}_{ki}) = \sum_{\mathbf{c}_k \sim \mathbf{s}_{ki}} \prod_{q,\, \mathbf{s}_{qk} \sim \mathbf{c}_k} M_{qk}(\mathbf{s}_{qk}) \prod_{\tau_k \sim \mathbf{c}_k} \tau_k,$$

while Method A gives:

$$dr(\mathbf{s}_{ki}) = \sum_{\mathbf{c}_k \sim \mathbf{s}_{ki}} dr(f) \prod_{q,\, \mathbf{s}_{qk} \sim \mathbf{c}_k} vr(\mathbf{s}_{qk}) \prod_{\tau_k \sim \mathbf{c}_k} \tau_k.$$

It then follows that $M_{ki}(\mathbf{s}_{ki}) = dr(\mathbf{s}_{ki})$ since $dr(f) = 1$ by description of Method A, and $M_{qk}(\mathbf{s}_{qk}) = vr(\mathbf{s}_{qk})$ by Part 1 (f is the root node in the arithmetic circuit). $\square$

Proof of Theorem 6

To prove this theorem, we need to distinguish between the values of registers r during the upward–pass and downward–pass of Method B, so we will use r^u and r^d to represent these values, respectively.

Part 1

We will show that after the inward–pass of Hugin, we have $\Phi_i(\mathbf{c}_i) = r^u(\mathbf{c}_i)$ and $\Phi_{ki}(\mathbf{s}_{ki}) = r^u(\mathbf{s}_{ki})$. But this is straightforward given the correspondence between the definition of a circuit embedded in a jointree, and given that Hugin initializes all separator entries to 1. Hence, the Hugin inward–pass is simply evaluating the circuit.

Part 2

Consider the same setup as in the proof of Theorem 5. Suppose that messages have been passed into k, and k is about to send a message to i. Assume by induction that $\Phi_k(\mathbf{c}_k) = r^d(\mathbf{c}_k)$. We want to show that after k sends its message into i, we have $\Phi_i(\mathbf{c}_i) = r^d(\mathbf{c}_i)$ and $\Phi_{ki}(\mathbf{s}_{ki}) = r^d(\mathbf{s}_{ki})$. Consider the following:

- before cluster k sends its message to i, we have $\Phi_i(\mathbf{c}_i) = r^u(\mathbf{c}_i)$ and $\Phi_{ki}(\mathbf{s}_{ki}) = r^u(\mathbf{s}_{ki})$ by Part 1;
- the correspondence between addition nodes $\mathbf{s}_{ki}$ and entries of separator table Φ_{ki};
- the correspondence between multiplication nodes $\mathbf{c}_i, \mathbf{c}_k$ and entries of cluster tables Φ_i, Φ_k;
- the correspondence between the processing of all addition nodes $\mathbf{s}_{ki}$ by Method B and the passing of message M_{ki} by Hugin propagation.

Hence, after passing the message form k to i, we must have $\Phi_i(\mathbf{c}_i) = r^d(\mathbf{c}_i)$ and $\Phi_{ki}(\mathbf{s}_{ki}) = r^d(\mathbf{s}_{ki})$.

The base case for this induction is when cluster k is the root. Note that the downward–pass of Method B does not change the registers of the root node f, neither does it change the registers of its children which correspond to the entries of the root cluster k. Hence, immediately after the upward–pass of Method B, we have $\Phi_k(\mathbf{c}_k) = r^d(\mathbf{c}_k) = r^u(\mathbf{c}_k)$. $\square$