Inference in Belief Networks: A Procedural Guide

Cecil Huang
Section on Medical Informatics
Stanford University School of Medicine

Adnan Darwiche [*]
Information Technology
Rockwell Science Center

Address correspondence to Cecil Huang, Section on Medical Informatics, MSOB X-215, Stanford, CA 94305-5479 USA

[*]Rockwell Science Center, 1049 Camino Dos Rios, Thousand Oaks, CA 91360 USA

International Journal of Approximate Reasoning 1994 11:1–158
© 1994 Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010 0888-613X/94/$7.00

ABSTRACT

Belief networks are popular tools for encoding uncertainty in expert systems. These networks rely on inference algorithms to compute beliefs in the context of observed evidence. One established method for exact inference on belief networks is the Probability Propagation in Trees of Clusters (PPTC) algorithm, as developed by Lauritzen and Spiegelhalter and refined by Jensen et al. [1, 2, 3] PPTC converts the belief network into a secondary structure, then computes probabilities by manipulating the secondary structure. In this document, we provide a self-contained, procedural guide to understanding and implementing PPTC. We synthesize various optimizations to PPTC that are scattered throughout the literature. We articulate undocumented, "open secrets" that are vital to producing a robust and efficient implementation of PPTC. We hope that this document makes probabilistic inference more accessible and affordable to those without extensive prior exposure.

Keywords: *Artificial intelligence, Bayesian network, belief network, causal network, evidence, expert systems, join tree, probabilistic inference, probability propagation, reasoning under uncertainty.*

1. INTRODUCTION

1.1. Purpose

An increasing number of academic and commercial endeavors use belief networks[1] to encode uncertain knowledge in complex domains. These networks rely on inference algorithms to compute beliefs of alternative hypotheses in the context of observed evidence. However, the task of realizing an inference algorithm is not trivial. Much effort is spent synthesizing methods that are scattered throughout the literature and converting them to algorithmic form. Additional effort is spent addressing undocumented, lower-level issues that are vital to producing a robust and efficient implementation. These issues exist, in the words of one colleague, as "open secrets" within the probabilistic inference community.

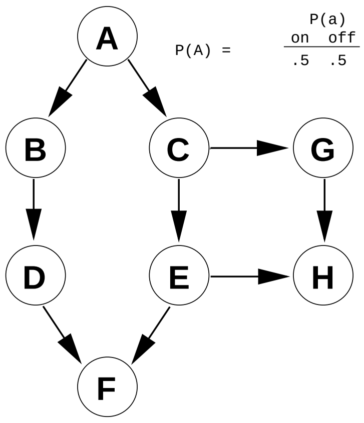
This document is addressed to interested researchers and developers who do not have extensive prior exposure to algorithms for probabilistic inference. We describe, in procedural fashion, the **Probability Propagation in Trees of Clusters** (PPTC) method for probabilistic inference, as developed by Lauritzen and Spiegelhalter and refined by Jensen et al. [1, 2, 3] We focus on the steps required to make PPTC work. We synthesize various published optimizations to PPTC, and we articulate the "open secrets" that are crucial to a robust and efficient implementation of PPTC. PPTC is an established method for *exact* probabilistic inference; other exact methods include cutset conditioning [6, 7, 8] and symbolic probabilistic inference (SPI) [9, 10]. A review of approximate methods can be found in [11].

Our goal is for the reader to be able to use this document to implement PPTC without additional help. We hope that this document makes probabilistic inference more accessible and affordable to those that are not entrenched in the belief networks community. More effort can then be spent conducting research and developing applications that make use of this technology.

1.2. What is PPTC?

PPTC is a method for performing probabilistic inference on a belief network. Consider the belief network shown in Figure 1. An example of probabilistic inference would be to compute the probability that $A = on$, given the knowledge that $C = on$ and $E = off$. In general, **probabilistic inference** on a belief network is the process of computing $P(V = v \mid \mathbf{E} = \mathbf{e})$, or simply $P(v \mid \mathbf{e})$, where v is a value of a variable V and $\mathbf{e}$ is an assignment of values to a set of variables $\mathbf{E}$ in the belief network. Basically, $P(v \mid \mathbf{e})$ asks:

[1]Belief networks are also referred to as *causal networks* and *Bayesian networks* in the literature. Comprehensive introductions to belief networks can be found in [4, 5].



P(A) =

P(a) on	off
.5	.5

P(B|A) =

a	P(b|a) on	off
on	.5	.5
off	.4	.6

P(C|A) =

a	P(c|a) on	off
on	.7	.3
off	.2	.8

P(D|B) =

a	P(d|b) on	off
on	.9	.1
off	.5	.5

P(E|C) =

a	P(e|c) on	off
on	.3	.7
off	.6	.4

P(F|DE) =

d	e	P(f|de) on	off
on	on	.01	.99
on	off	.01	.99
off	on	.01	.99
off	off	.99	.01

P(G|C) =

c	P(g|c) on	off
on	.8	.2
off	.1	.9

P(H|EG) =

g	h	P(h|eg) on	off
on	on	.05	.95
on	off	.95	.05
off	on	.95	.05
off	off	.95	.05

Figure 1. A belief network.

Suppose that I observe **e** on a set of variables **E**, what is the probability that the variable V has value v, given **e**?

PPTC works in two steps. First, a belief network is converted into a secondary structure. Then, probabilities of interest are computed by operating on that secondary structure.

1.3. Overview Of Document

In Section 2, we describe notational conventions and fundamental concepts that are used throughout the document. Then in Section 3, we introduce belief networks and their secondary structures. In Sections 4 and 5 we describe the creation of the secondary structure, beginning with the belief network. We integrate evidence into the above framework in Section 6. These sections constitute the essence of PPTC inference. Having laid these foundations, we discuss some optimization opportunities in Section 7 and low-level implementation issues in Section 8.

2. NOTATION

We specify PPTC using the following notational conventions and fundamental concepts.

2.1. Variables And Values

We denote **variables** with italic uppercase letters (A, B, C), and variable **values** with italic lowercase letters (a, b, c). We **instantiate** a variable A by assigning it a value a; we call a an **instantiation** of A.

Sets of variables are denoted by boldface uppercase letters ($\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$), and their instantiations by boldface lowercase letters ($\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$). We instantiate a set of variables $\mathbf{X}$ by assigning a value to each variable in $\mathbf{X}$; we denote this assignment with $\mathbf{x}$, and call $\mathbf{x}$ an instantiation of $\mathbf{X}$.

2.2. Potentials And Distributions

2.2.1. Potentials We define a **potential** [1] over a set of variables $\mathbf{X}$ as a function that maps each instantiation $\mathbf{x}$ into a nonnegative real number; we denote this potential as $\phi_{\mathbf{X}}$. We use the notation $\phi_{\mathbf{X}}(\mathbf{x})$ to denote the number that $\phi_{\mathbf{X}}$ maps $\mathbf{x}$ into; we call $\phi_{\mathbf{X}}(\mathbf{x})$ an **element**. Potentials can be viewed as matrices and implemented as tables, so we will also refer to them as **matrices** and **tables**.

2.2.2. Operations On Potentials We define two basic operations on potentials: **marginalization** and **multiplication** [2]. Suppose we have a set of variables $\mathbf{Y}$, its potential $\phi_{\mathbf{Y}}$, and a set of variables $\mathbf{X}$ where $\mathbf{X} \subseteq \mathbf{Y}$. The **marginalization** of $\phi_{\mathbf{Y}}$ into $\mathbf{X}$ is a potential $\phi_{\mathbf{X}}$, where each $\phi_{\mathbf{X}}(\mathbf{x})$ is computed as follows:

1. Identify the instantiations $\mathbf{y_1}, \mathbf{y_2}, \ldots$ that are consistent with $\mathbf{x}$.
2. Assign to $\phi_{\mathbf{X}}(\mathbf{x})$ the sum $\phi_{\mathbf{Y}}(\mathbf{y_1}) + \phi_{\mathbf{Y}}(\mathbf{y_2}) + \ldots$

This marginalization is denoted as follows:

$$\phi_{\mathbf{X}} = \sum_{\mathbf{Y} \setminus \mathbf{X}} \phi_{\mathbf{Y}}.$$

Given two sets of variables $\mathbf{X}$ and $\mathbf{Y}$ and their potentials $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$, the **multiplication** of $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$ is a potential $\phi_{\mathbf{Z}}$, where $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$, and each $\phi_{\mathbf{Z}}(\mathbf{z})$ is computed as follows:

1. Identify the instantiation $\mathbf{x}$ and the instantiation $\mathbf{y}$ that are consistent with $\mathbf{z}$.

2. Assign to $\phi_{\mathbf{Z}}(\mathbf{z})$ the product $\phi_{\mathbf{X}}(\mathbf{x})\phi_{\mathbf{Y}}(\mathbf{y})$.

This multiplication of potentials is denoted as follows:

$$\phi_{\mathbf{Z}} = \phi_{\mathbf{X}}\phi_{\mathbf{Y}}.$$

2.2.3. Probability Distributions A **probability distribution**, or simply a **distribution**, is a special case of a potential. Given a set of variables $\mathbf{X}$, we use the notation $P(\mathbf{X})$ to denote the **probability distribution of X**, or simply the **probability of X**. $P(\mathbf{X})$ is a potential over $\mathbf{X}$ whose elements add up to 1. We denote the elements of $P(\mathbf{X})$ as $P(\mathbf{x})$, and we call each element $P(\mathbf{x})$ the **probability of x**. With this notation, we have

$$\sum_{\mathbf{x}} P(\mathbf{x}) = 1.$$

Another important notion is that of conditional probability. Given sets of variables $\mathbf{X}$ and $\mathbf{Y}$, we use the notation $P(\mathbf{X} \mid \mathbf{Y})$ to denote the **conditional probability of X given Y**, or simply the **probability of X given Y**. $P(\mathbf{X} \mid \mathbf{Y})$ is a collection of probability distributions indexed by the instantiations of $\mathbf{Y}$; each $P(\mathbf{X} \mid \mathbf{y})$ is a probability distribution over $\mathbf{X}$. We denote the elements of $P(\mathbf{X} \mid \mathbf{y})$ as $P(\mathbf{x} \mid \mathbf{y})$, and we call each element $P(\mathbf{x} \mid \mathbf{y})$ the **probability of x given y**. With this notation, we have, for each instantiation $\mathbf{y}$,

$$\sum_{\mathbf{x}} P(\mathbf{x} \mid \mathbf{y}) = 1.$$

Note that $P(\mathbf{X})$ is a special case of $P(\mathbf{X} \mid \mathbf{Y})$ where $\mathbf{Y} = \emptyset$.

3. BELIEF NETWORKS AND THEIR SECONDARY STRUCTURES

3.1. Belief Networks

Belief networks are used by experts to encode selected aspects of their knowledge and beliefs about a domain. Once constructed, the network induces a probability distribution over its variables.

3.1.1. Definition A belief network over a set of variables $\mathbf{U} = \{V_1, \ldots, V_n\}$ consists of two components:

- A **directed acyclic graph (DAG)** $\mathcal{G}$: Each vertex in the graph represents a variable V, which takes on values v_1, v_2, etc.[2] The **parents** of V in the graph are denoted by $\mathbf{\Pi}_V$, with instantiations π_V; the **family** of V, denoted by $\mathbf{F_V}$, is defined as $\{V\} \cup \mathbf{\Pi}_V$. The DAG structure encodes a set of **independence assertions**, which restrict the variety of interactions that can occur among variables. These assertions are discussed more precisely in Section 3.1.3 below.

- A **quantification** of $\mathcal{G}$: Each variable in $\mathcal{G}$ is quantified with a conditional probability table $P(V \mid \mathbf{\Pi}_V)$. While $P(V \mid \mathbf{\Pi}_V)$ is technically a function of $\mathbf{F_V}$, it is most helpful to think of it in the following way: for each instantiation π_V, real numbers in $[0,1]$ are assigned to each value v, such that they add up to 1. When $\mathbf{\Pi}_V \neq \emptyset$, $P(V \mid \mathbf{\Pi}_V)$ is called the **conditional probability of V given $\mathbf{\Pi}_V$**; when $\mathbf{\Pi}_V = \emptyset$, $P(V \mid \mathbf{\Pi}_V)$, or simply $P(V)$, is called the **prior probability** of V.

These components induce a **joint probability distribution** over $\mathbf{U}$, given by

$$P(\mathbf{U}) = \prod_{i=1}^{n} P(V_i \mid \mathbf{C}_{V_i}),$$

where $V_1, \ldots, V_n$ are the variables in the network.

3.1.2. Example Refer to the example belief network shown in Figure 1. This network is over the set of variables $\mathbf{U} = \{A, B, C, D, E, F, G, H\}$, each variable having values $\{on, off\}$. $P(F \mid DE)$ is an example of a conditional probability; $P(A)$ is an example of a prior probability. The network's joint probability distribution is the product of the conditional and prior probabilities:

$$P(\mathbf{U}) = P(A)\,P(B \mid A)\,P(C \mid A)\,P(D \mid B)\,P(E \mid C)\,P(F \mid DE)\,P(G \mid C)\,P(H \mid EG).$$

[2] We will not distinguish between a vertex and the variable it represents.

3.1.3. Independence Assertions In addition to the numbers in the tables, a belief network also encodes independence assertions, which do not depend on how the network is quantified. An independence assertion is a statement of the form **X and Y are independent given Z**: for all combinations of values $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, $P(\mathbf{x} \mid \mathbf{z}) = P(\mathbf{x} \mid \mathbf{yz})$.[3] In other words, if we are given $\mathbf{z}$, then knowing $\mathbf{y}$ would not affect our belief in $\mathbf{x}$. The independence assertions in a belief network are important because PPTC uses them to reduce the complexity of inference.

The pattern of arcs in the DAG encodes the following independence assertions: each variable is independent of its nondescendants, given its parents. Two or more independence assertions can logically imply a new independence assertion, using a mechanism of manipulating such statements known as the graphoid axioms [12]. A graph-theoretic relation known as **d-separation** captures *all* such derivable independences encoded by the DAG [13]. In other words, $\mathbf{Z}$ d-separates $\mathbf{X}$ and $\mathbf{Y}$ in the DAG iff, in the network, $\mathbf{X}$ and $\mathbf{Y}$ are independent given $\mathbf{Z}$, with respect to the graphoid axioms.[4]

3.2. The Secondary Structure

While experts typically use belief networks to encode their domain, PPTC performs probabilistic inference on a secondary structure that we characterize in Section 3.2.1 below.

3.2.1. Definition Given a belief network over a set of variables $\mathbf{U} = \{V_1, \ldots, V_n\}$, we define a secondary structure that contains a graphical and a numerical component. The graphical component consists of the following:

- An undirected **tree** $\mathcal{T}$: Each node in $\mathcal{T}$ is a **cluster** (nonempty set) of variables. The clusters satisfy the **join tree property**: given two clusters $\mathbf{X}$ and $\mathbf{Y}$ in $\mathcal{T}$, all clusters on the path between $\mathbf{X}$ and $\mathbf{Y}$ contain $\mathbf{X} \cap \mathbf{Y}$.[5] For each variable $V \in \mathbf{U}$, the family of V, $\mathbf{F_V}$ (Section 3.1.1), is included in at least one of the clusters.

- **Sepsets**: Each edge in $\mathcal{T}$ is labeled with the intersection of the adjacent clusters; these labels are called separator sets, or sepsets.[6]

The numerical component is described using the notion of a **belief potential**. A belief potential is a function that maps each instantiation of a set of variables into a real number (Section 2.2.1). Belief potentials are defined over the following sets of variables:

[3] Or, equivalently, $P(\mathbf{xy} \mid \mathbf{z}) = P(\mathbf{x} \mid \mathbf{z})P(\mathbf{y} \mid \mathbf{z})$.
[4] An intuitive discussion on d-separation can be found in [14].
[5] We will not distinguish between a cluster and its variables.
[6] Note that if a sepset is included as a cluster, the resulting cluster tree would still satisfy the join tree property.

- Clusters: Each cluster $\mathbf{X}$ is associated with a belief potential $\phi_{\mathbf{X}}$ that maps each instantiation $\mathbf{x}$ into a real number.
- Sepsets: Each sepset $\mathbf{S}$ is associated with a belief potential $\phi_{\mathbf{S}}$ that maps each instantiation $\mathbf{s}$ into a real number.

The belief potentials are not arbitrarily specified; they must satisfy the following constraints:

- For each cluster $\mathbf{X}$ and neighboring sepset $\mathbf{S}$, it holds that

  $$\sum_{\mathbf{X}\setminus\mathbf{S}} \phi_{\mathbf{X}} = \phi_{\mathbf{S}}. \tag{1}$$

  When Equation 1 is satisfied for a cluster $\mathbf{X}$ and neighboring sepset $\mathbf{S}$, we say that $\phi_{\mathbf{S}}$ is **consistent** with $\phi_{\mathbf{X}}$. When consistency holds for every cluster-sepset pair, we say that the secondary structure is **locally consistent**.

- The belief potentials encode the joint distribution $P(\mathbf{U})$ of the belief network according to

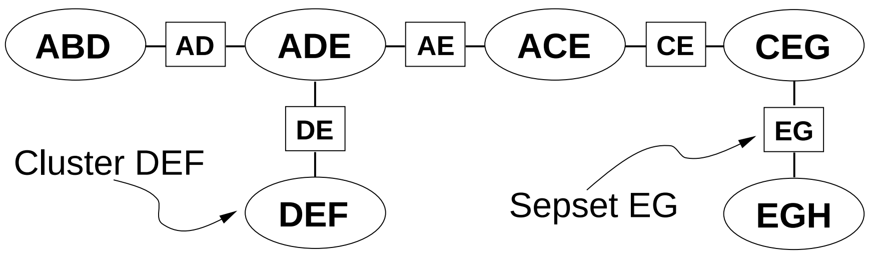
  $$P(\mathbf{U}) = \frac{\prod_i \phi_{\mathbf{X_i}}}{\prod_j \phi_{\mathbf{S_j}}}, \tag{2}$$

  where $\phi_{\mathbf{X_i}}$ and $\phi_{\mathbf{S_j}}$ are the cluster and sepset potentials, respectively.

A key step in PPTC is the construction of a secondary structure that satisfies the above constraints. Such a secondary structure has the following important property: for each cluster (or sepset) $\mathbf{X}$, it holds that $\phi_{\mathbf{X}} = P(\mathbf{X})$ [2]. Using this property, we can compute the probability distribution of any variable V, using any cluster (or sepset) $\mathbf{X}$ that contains V, as follows:

$$P(V) = \sum_{\mathbf{X}\setminus\{\mathbf{V}\}} \phi_{\mathbf{X}}. \tag{3}$$

The secondary structure has been referred to in the literature as a *join tree*, *junction tree*, *tree of belief universes*, *cluster tree*, and *clique tree*, among other designations. In this document, we use the term **join tree** to refer to the graphical component, and the term **join tree potential** to refer generically to a cluster or sepset belief potential. We will also use the term *join tree* to refer to the entire secondary structure, as it is being created; the meaning of *join tree* will be clear from the context. In Section 4, we show how to build a join tree from the DAG of a belief network, and in Section 5, we describe how PPTC manipulates the join tree potentials so that they satisfy Equations (1) and (2).



ϕ_{ABD} =

a	b	d	ϕ_{ABD}(abd)
on	on	on	.225
on	on	off	.025
on	off	on	.125
on	off	off	.125
off	on	on	.180
off	on	off	.020
off	off	on	.150
off	off	off	.150

ϕ_{AD} =

a	d	ϕ_{AD}(ad)
on	on	.35
on	off	.15
off	on	.33
off	off	.17

etc.

Figure 1. An example of a secondary structure.

3.2.2. Example Figure 1 illustrates part of a secondary structure obtained from the belief network in Figure 1. The tree contains clusters $\{ABD, ACE, ADE, CEG, DEF, EGH\}$ and sepsets $\{AD, AE, CE, DE, EG\}$, each with a belief potential ϕ over its variables. For example, ϕ_{ABD} and ϕ_{AD} are illustrated in Figure 1. Note that ϕ_{ABD} and ϕ_{AD} satisfy the local consistency requirement, since we have

$$\phi_{AD} = \sum_{B} \phi_{ABD}.$$

Local consistency also holds for the other cluster-sepset pairs. Finally, the belief potentials encode the joint distribution of the belief network by satisfying

$$P(\mathbf{U}) \; = \frac{\phi_{ABD}\,\phi_{ACE}\,\phi_{ADE}\,\phi_{CEG}\,\phi_{DEF}\,\phi_{EGH}}{\phi_{AD}\,\phi_{AE}\,\phi_{CE}\,\phi_{DE}\,\phi_{EG}}.$$

3.2.3. Independence Assertions The complete set of independence assertions encoded by the join tree can be specified as follows [15]. Begin with a secondary structure over the set of variables $\mathbf{U}$, in which the sepsets are included as clusters. Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ be subsets of $\mathbf{U}$. The tree shows $\mathbf{X}$ to be independent of $\mathbf{Y}$ given $\mathbf{Z}$ if, for each $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$, the chain between any cluster containing X and any cluster containing Y passes through a cluster $\mathbf{Z}$.

4. BUILDING JOIN TREES FROM BELIEF NETWORKS

In this section, we begin with the DAG of a belief network, and apply a series of graphical transformations that result in a join tree. These transformations involve a number of intermediate structures, and can be summarized as follows:

1. Construct an undirected graph, called a **moral graph**, from the DAG.
2. Selectively add arcs to the moral graph to form a **triangulated graph**.
3. From the triangulated graph, identify select subsets of nodes, called **cliques**.
4. Build a join tree, starting with the cliques as clusters: connect the clusters to form an undirected tree satisfying the join tree property, inserting the appropriate sepsets.

Steps 2 and 4 are nondeterministic; consequently, many different join trees can be built from the same DAG.

4.1. Constructing The Moral Graph

Let $\mathcal{G}$ be the DAG of a belief network. The **moral graph** $\mathcal{G}_M$ corresponding to $\mathcal{G}$ is constructed as follows [1, 16]:

1. Create the undirected graph $\mathcal{G}_u$ by copying $\mathcal{G}$, but dropping the directions of the arcs.
2. Create $\mathcal{G}_M$ from $\mathcal{G}_u$ as follows: For each node V, identify its parents $\mathbf{\Pi}_V$ in $\mathcal{G}$. Connect each pair of nodes in $\mathbf{\Pi}_V$ by adding undirected arcs to $\mathcal{G}_u$.

Figure 1 illustrates this transformation on the DAG from Figure 1. The undirected arcs added to $\mathcal{G}_u$ are called **moral arcs**, shown as dashed lines in the figure.

4.2. Triangulating The Moral Graph

An undirected graph is **triangulated** iff every cycle of length four or greater contains an edge that connects two nonadjacent nodes in the cycle. We describe a procedure for triangulating an arbitrary undirected graph, adapted from Kjærulff [17]:

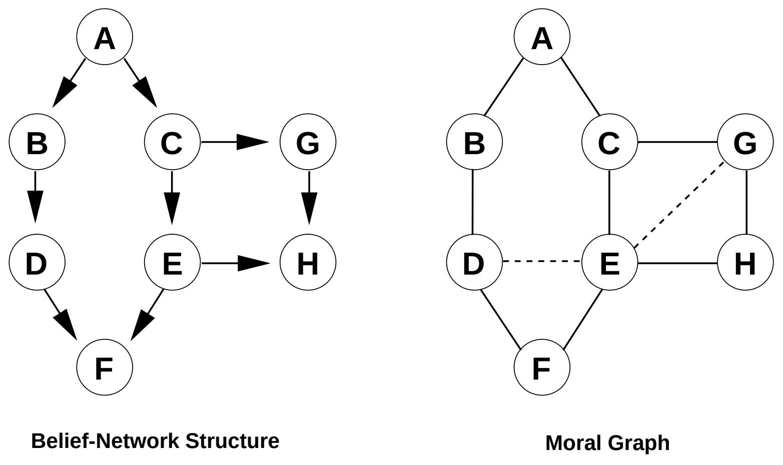


Figure 1. Constructing the moral graph.

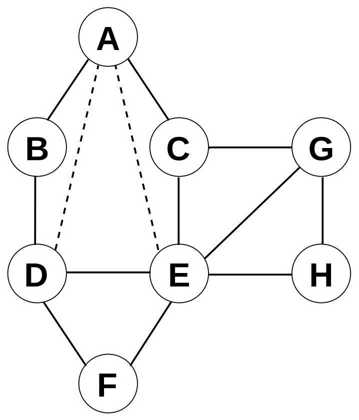
1. Make a copy of $\mathcal{G}_M$; call it $\mathcal{G}'_M$.
2. While there are still nodes left in $\mathcal{G}'_M$:
   - **(a)** Select a node V from $\mathcal{G}'_M$, according to the criterion described below.
   - **(b)** The node V and its neighbors in $\mathcal{G}'_M$ form a **cluster**. Connect all of the nodes in this cluster. For each edge added to $\mathcal{G}'_M$, add the same corresponding edge to $\mathcal{G}_M$.
   - **(c)** Remove V from $\mathcal{G}'_M$.
3. $\mathcal{G}_M$, modified by the additional arcs introduced in the previous steps, is now triangulated.

To describe the criterion for selecting the nodes in Step 2a, we rely on the following notion of a weight:

- The **weight** of a node V is the number of values of V.
- The **weight** of a cluster is the product of the weights of its constituent nodes.

The criterion for selecting nodes to remove is now stated as follows: *Choose the node that causes the least number of edges to be added in Step 2b, breaking ties by choosing the node that induces the cluster with the smallest weight.*[7]

[7] We access the next node to be removed by keeping the remaining nodes of $\mathcal{G}'_M$ in a **binary heap**. Each node V is associated with a primary key (the number of edges



Eliminated Vertex	Induced Cluster	Edges Added
H	EGH	none
G	CEG	none
F	DEF	none
C	ACE	(A,E)
B	ABD	(A,D)
D	ADE	none
E	AE	none
A	A	none

Elimination Ordering

Figure 2. Triangulating the moral graph.

Figure 2 depicts the triangulated graph, as obtained from the moral graph in Figure 1. The dashed lines in the figure indicate the edges added to triangulate the moral graph. We also show the **elimination ordering** of the nodes, so that the interested reader can trace each step in the triangulation process.

In general, there are many ways to triangulate an undirected graph. An optimal triangulation is one that minimizes the sum of the state space sizes of the cliques (Section 4.3) of the triangulated graph. The task of finding an optimal triangulation is $\mathcal{NP}$-complete [19]. However, the node-selection criterion in Step 2a is a greedy, polynomial-time *heuristic* that produces high-quality triangulations in real-world settings [17].

4.3. Identifying Cliques

A **clique** in an undirected graph $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is complete and maximal. **Complete** means that every pair of distinct nodes is connected by an edge. **Maximal** means that the clique is not properly contained in a larger, complete subgraph. Golumbic [20] offers an efficient algorithm for identifying the cliques of an arbitrary triangulated graph;

added if V were selected next) and a secondary key (the weight of the cluster induced if V were selected next). When V is removed, each of V's neighbors needs to have its keys recalculated, and, therefore, its position in the heap modified. Removing a node V costs $O(k \lg n)$ time, where k is the number of neighbors of V in $\mathcal{G}'_M$, and n is the number of nodes remaining in $\mathcal{G}'_M$. A more detailed discussion on binary heaps can be found in [18].

this algorithm relies on a particular ordering of the nodes, which can be generated according to Tarjan and Yannakakis [21].

By adapting the triangulation procedure in Section 4.2, though, we can identify the cliques of the triangulated graph as it is being constructed. Our procedure relies on the following two observations:

- Each clique in the triangulated graph is an induced cluster from step 2b of Section 4.2.
- An induced cluster can never be a subset of a subsequently induced cluster.

These observations suggest that we can extract the cliques during the triangulation process by *saving each induced cluster that is not a subset of any previously saved cluster*. Revisiting Figure 2, we see that the cliques of the triangulated graph are EGH, CEG, DEF, ACE, ABD, and ADE.

4.4. Building an optimal join tree

From this point on, we no longer need the undirected graph. We seek to build an optimal join tree by connecting the cliques obtained in Section 4.3 above.[8] To build an optimal join tree, we must connect the cliques so that the resulting clique tree satisfies the join tree property and an optimality criterion that we will define below. The join tree property is essential for the tree to be useful for probabilistic inference, and the optimality criterion favors those join trees that minimize the computational time required for inference.

Given a set of n cliques, we can form a clique tree by iteratively inserting edges between pairs of cliques, until the cliques are connected by $n-1$ edges. We can also view this task as iteratively inserting sepsets between pairs of cliques, until the cliques are connected by $n-1$ sepsets [22]. We take this latter approach in specifying how to build an optimal join tree. We divide our specification of the algorithm into two parts: First, in Section 4.4.1, we provide a generic procedure that forms a clique tree by iteratively selecting and inserting candidate sepsets. Then, in Section 4.4.2, we show how the sepsets must be chosen, in order for the clique tree to be an optimal join tree.

4.4.1. Forming The Clique Tree The following procedure builds an optimal join tree by iteratively selecting and inserting candidate sepsets [22]; the criterion in Step 3a is specified later in Section 4.4.2 below.

[8]The cliques of the triangulated graph will become the clusters of the join tree; hence, we will use the terms *clique* and *cluster* interchangeably in this section. However, in general, a join tree need not be a clique tree.

Building an Optimal Join Tree

1. Begin with a set of n trees, each consisting of a single clique, and an empty set $\mathcal{S}$.

2. For each distinct pair of cliques $\mathbf{X}$ and $\mathbf{Y}$:[9]

 (a) Create a candidate sepset, labeled $\mathbf{X} \cap \mathbf{Y}$, with backpointers to the cliques $\mathbf{X}$ and $\mathbf{Y}$. Refer to this sepset as $\mathbf{S_{XY}}$.
 (b) Insert $\mathbf{S_{XY}}$ into $\mathcal{S}$.

3. Repeat until $n - 1$ sepsets have been inserted into the forest.

 (a) Select a sepset $\mathbf{S_{XY}}$ from $\mathcal{S}$, according to the criterion specified in Section 4.4.2. Delete $\mathbf{S_{XY}}$ from $\mathcal{S}$.
 (b) Insert the sepset $\mathbf{S_{XY}}$ between the cliques $\mathbf{X}$ and $\mathbf{Y}$ *only if* $\mathbf{X}$ and $\mathbf{Y}$ are on different trees in the forest.[10] (Note that the insertion of such a sepset would merge two trees into a larger tree.)

4.4.2. Choosing The Appropriate Sepsets In order to describe how to choose the next candidate sepset, we define the notions of **mass** and **cost**, as follows:

- The **mass** of a sepset $\mathbf{S_{XY}}$ is the number of variables it contains, or the number of variables in $\mathbf{X} \cap \mathbf{Y}$.

- The **cost** of a sepset $\mathbf{S_{XY}}$ is the weight of $\mathbf{X}$ plus the weight of $\mathbf{Y}$, where *weight* is defined as follows:

  – The **weight** of a variable V is the number of values of V.
  – The **weight** of a set of variables $\mathbf{X}$ is the product of the weights of the variables in $\mathbf{X}$.

With these notions established, we can now state how to select the next candidate sepset from $\mathcal{S}$ whenever we execute Step 3a in Section 4.4.1 [22]:

- For the resulting clique tree to satisfy the join tree property, we must *choose the candidate sepset with the largest mass.*

- When two or more sepsets of equal mass can be chosen, we can optimize the inference time on the resulting join tree by breaking the tie as follows: *choose the candidate sepset with the smallest cost.*

The basis for this method of building an optimal join tree can be found in [22].

[9] There will be $n(n-1)/2$ such pairs.
[10] Otherwise, a cycle would form.

4.4.3. Example Starting with the clique set $\{ABD, ACE, ADE, CEG, DEF, EGH\}$ from Figure 2, we choose the connecting sepsets AD, AE, CE, DE, and EG based on their mass. These cliques and sepsets form the join tree structure illustrated in Figure 1.

4.4.4. Implementation Notes Similar to the triangulation algorithm, we can implement the set of candidate sepsets $\mathcal{S}$ as a binary heap, ranking each sepset according to a primary key (mass) and a secondary key (cost).

Note that some of the candidate sepsets in Step 2a of Section 4.4.1 are empty. If the original DAG of the belief network is not fully connected, then some of these empty candidate sepsets will be included in the final join tree. This outcome is acceptable; however, one optimization involves disallowing empty sepsets, and terminating Step 3 when $n-1$ sepsets have been chosen, or when $\mathcal{S}$ is empty. If Step 3 terminates because of the latter case alone, the resulting join tree will actually be a join *forest*. Section 7.2 discusses how to deal with such forests.

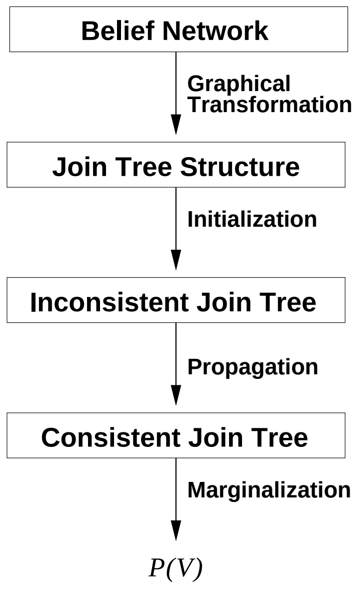


Figure 1. Block diagram of PPTC with no evidence.

5. PRINCIPLES OF INFERENCE

Having built a join tree structure, we now provide procedures for computing the join tree's numerical component, so that it satisfies the conditions in Section 3.2.1. We show how to compute the probability distribution $P(V)$, for any variable V, using this join tree. Note that computing $P(V)$ corresponds to probabilistic inference in the context of no evidence. We address the more general problem of computing $P(V \mid \mathbf{e})$, in the context of evidence $\mathbf{e}$, later in Section 6.

5.1. Overview

Figure 1 illustrates the overall control for PPTC with no evidence. We trace the steps in this figure as follows:

- *Graphical Transformation.* Transform the DAG of a *belief network* into a *join tree structure*, using the procedures in Section 4 above.
- *Initialization* (Section 5.2). Quantify the join tree with belief potentials so that they satisfy Equation (2). The result is an *inconsistent join tree*, as this initial assignment of belief potentials does not meet the local consistency requirements of Equation (1).

- *Global Propagation* (Section 5.3). Perform an ordered series of local manipulations, called **message passes**, on the join tree potentials. The message passes rearrange the join tree potentials so that they become locally consistent; thus, the result of global propagation is a *consistent join tree*, which satisfies both Equations (1) and (2).

- *Marginalization* (Section 5.4). From the consistent join tree, compute $P(V)$ for each variable of interest V.

5.2. Initialization

The following procedure assigns initial join tree potentials, using the conditional probabilities from the belief network:

1. For each cluster and sepset $\mathbf{X}$, set each $\phi_{\mathbf{X}}(\mathbf{x})$ to 1:

$$\phi_{\mathbf{X}} \longleftarrow 1.$$

2. For each variable V, perform the following: Assign to V a cluster $\mathbf{X}$ that contains $\mathbf{F_V}$;[11] call $\mathbf{X}$ the **parent cluster of $\mathbf{F_V}$**. Multiply $\phi_{\mathbf{X}}$ by $P(V \mid \mathbf{\Pi}_V)$:

$$\phi_{\mathbf{X}} \longleftarrow \phi_{\mathbf{X}} P(V \mid \mathbf{\Pi}_V).$$

After initialization, the conditional distribution $P(V \mid \mathbf{\Pi}_V)$ of each variable V has been multiplied into some cluster potential. The initialization procedure satisfies Equation (2) as follows:

$$\frac{\prod_{i=1}^{N} \phi_{\mathbf{X_i}}}{\prod_{j=1}^{N-1} \phi_{\mathbf{S_j}}} = \frac{\prod_{k=1}^{Q} P(V_k \mid \mathbf{C}_{V_k})}{1} = P(\mathbf{U}),$$

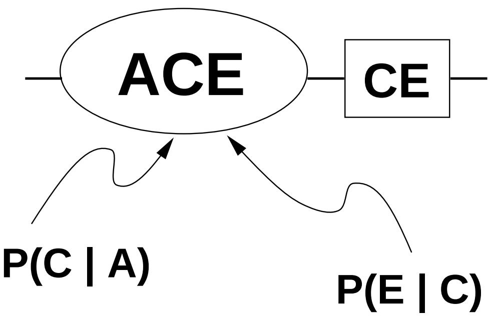
where N is the number of clusters, Q is the number of variables, and $\phi_{\mathbf{X_i}}$ and $\phi_{\mathbf{S_j}}$ are the cluster and sepset potentials, respectively.

Figure 2 illustrates the initialization procedure on the tables of cluster ACE and sepset CE from the secondary structure of Figure 1. In this example, ACE is the parent cluster of $\mathbf{F_C}$ and $\mathbf{F_E}$, but not $\mathbf{F_A}$. Thus, after initialization, $\phi_{ACE} = P(C \mid A)P(E \mid C)$, and $\phi_{CE} = 1$.

5.3. Global Propagation

Having initialized the join tree potentials, we now perform global propagation in order to make them locally consistent. Global propagation consists of a series of local manipulations, called **message passes**, that occur

[11]The triangulation procedure in Section 4.2 guarantees that such a cluster exists.



$\boxed{\phi_{ACE}}$

a	c	e	Initial Values
on	on	on	1 × .7 × .3 = .21
on	on	off	1 × .7 × .7 = .49
on	off	on	1 × .3 × .6 = .18
on	off	off	1 × .3 × .4 = .12
off	on	on	1 × .2 × .3 = .06
off	on	off	1 × .2 × .7 = .14
off	off	on	1 × .8 × .6 = .48
off	off	off	1 × .8 × .4 = .32

$\boxed{\phi_{CE}}$

c	e	Initial Values
on	on	1
on	off	1
off	on	1
off	off	1

etc.

Figure 2. Initialization of cluster ACE and sepset CE.

between a cluster **X** and a neighboring cluster **Y**. A message pass from **X** to **Y** forces the belief potential of the intervening sepset to be consistent with $\phi_{\mathbf{X}}$ (see Equation (1)), while preserving the invariance of Equation (2). Global propagation causes each cluster to pass a message to each of its neighbors; these message passes are ordered so that each message pass will preserve the consistency introduced by previous message passes. When global propagation is completed, each cluster-sepset pair is consistent, and the join tree is locally consistent.

In Section 5.3.1, we describe a single message pass between two adjacent clusters. Then in Section 5.3.2, we explain how global propagation achieves local consistency by coordinating multiple message passes.

5.3.1. Single Message Pass Consider two adjacent clusters **X** and **Y** with sepset **R**, and their associated belief potentials $\phi_{\mathbf{X}}$, $\phi_{\mathbf{Y}}$, and $\phi_{\mathbf{R}}$. A **message pass from X to Y** occurs in two steps:

1. Projection. Assign a new table to **R**, saving the old table:

$$\phi_{\mathbf{R}}^{old} \longleftarrow \phi_{\mathbf{R}}.$$

$$\phi_{\mathbf{R}} \quad \longleftarrow \quad \sum_{\mathbf{X}\setminus\mathbf{R}} \phi_{\mathbf{X}}. \tag{1}$$

2. **Absorption**. Assign a new table to $\mathbf{Y}$, using both the old and the new tables of $\mathbf{R}$:

$$\phi_{\mathbf{Y}} \longleftarrow \phi_{\mathbf{Y}} \frac{\phi_{\mathbf{R}}}{\phi_{\mathbf{R}}^{old}}. \tag{2}$$

   For any instantiation $\mathbf{r}$ of $\mathbf{R}$, Jensen [23] shows that $\phi_{\mathbf{R}}^{old}(\mathbf{r}) = 0$ only if $\phi_{\mathbf{R}}(\mathbf{r}) = 0$. Whenever this occurs, set $0/0 = 0$.

Equations (1) and (2) assign new potentials to $\mathbf{R}$ and $\mathbf{Y}$; however, the left-hand side of Equation (2) remains constant, thus preserving the invariance of Equation (2):

$$\left(\frac{\prod_i \phi_{\mathbf{X_i}}}{\prod_j \phi_{\mathbf{S_j}}}\right) \frac{\phi_{\mathbf{R}}^{old}}{\phi_{\mathbf{R}}} \frac{\phi_{\mathbf{Y}}}{\phi_{\mathbf{Y}}^{old}} = \left(\frac{\prod_i \phi_{\mathbf{X_i}}}{\prod_j \phi_{\mathbf{S_j}}}\right) \frac{\phi_{\mathbf{R}}^{old}}{\phi_{\mathbf{R}}} \frac{\phi_{\mathbf{Y}}^{old} \frac{\phi_{\mathbf{R}}}{\phi_{\mathbf{R}}^{old}}}{\phi_{\mathbf{Y}}^{old}} = P(\mathbf{U}).$$

5.3.2. Coordinating Multiple Messages Given a join tree with n clusters, the PPTC global propagation algorithm begins by choosing an arbitary cluster $\mathbf{X}$, and then performing $2(n-1)$ message passes, divided into two phases. During the Collect-Evidence phase, each cluster passes a message to its neighbor in $\mathbf{X}$'s direction, beginning with the clusters farthest from $\mathbf{X}$. During the Distribute-Evidence phase, each cluster passes messages to its neighbors *away* from $\mathbf{X}$'s direction, beginning with $\mathbf{X}$ itself. The Collect-Evidence phase causes $n-1$ messages to be passed, and the Distribute-Evidence phase causes another $n-1$ messages to be passed.

Global propagation

1. Choose an arbitrary cluster $\mathbf{X}$.
2. Unmark all clusters. Call Collect-Evidence($\mathbf{X}$).
3. Unmark all clusters. Call Distribute-Evidence($\mathbf{X}$).

COLLECT-EVIDENCE(X)

1. Mark $\mathbf{X}$.
2. Call Collect-Evidence recursively on $\mathbf{X}$'s unmarked neighboring clusters, if any.
3. Pass a message from $\mathbf{X}$ to the cluster which invoked Collect-Evidence($\mathbf{X}$).

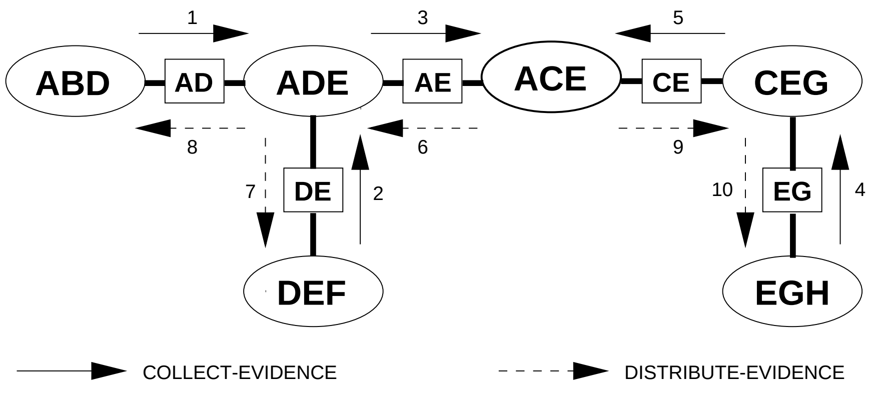


Figure 3. Message passing during global propagation.

DISTRIBUTE-EVIDENCE(X)

1. Mark **X**.
2. Pass a message from **X** to each of its unmarked neighboring clusters, if any.
3. Call DISTRIBUTE-EVIDENCE recursively on **X**'s unmarked neighboring clusters, if any.

The net result of this message passing is that each cluster passes its information, as encoded in its belief potential, to all of the other clusters in the join tree. Note that in this message-passing scheme, *a cluster passes a message to a neighbor only after it has received messages from all of its other neighbors.* This condition assures local consistency of the join tree when global propagation is completed [2, 23].

5.3.3. Example Figure 3 illustrates the PPTC propagation step on the join tree from Figure 1. Here, ACE is the starting cluster. During the COLLECT-EVIDENCE phase, messages are passed in ACE's direction, beginning with the clusters ABD, DEF, and EGH; these messages are indicated by the solid arrows. During the DISTRIBUTE-EVIDENCE phase, messages are passed *away* from cluster ACE, beginning with ACE; these messages are indicated by the dashed arrows. The numbers indicate one possible message passing order.

5.4. Marginalization

Once we have a consistent join tree, we can compute $P(V)$ for each variable of interest V as follows:

ϕ_{ABD} =

a	b	d	ϕ_{ABD}(abd)
on	on	on	.225
on	on	off	.025
on	off	on	.125
on	off	off	.125
off	on	on	.180
off	on	off	.020
off	off	on	.150
off	off	off	.150

$P(A) = \sum_{BD} \phi_{ABD} =$

a	P(a)
on	.225 + .025 + .125 + .125 = .500
off	.180 + .020 + .150 + .150 = .500

$P(D) = \sum_{AB} \phi_{ABD} =$

d	P(d)
on	.225 + .125 + .180 + .150 = .680
off	.025 + .125 + .020 + .150 = .320

Figure 4. Marginalization example.

1. Identify a cluster (or sepset) $\mathbf{X}$ that contains V.[12]

2. Compute $P(V)$ by marginalizing $\phi_{\mathbf{X}}$ according to Equation (3), repeated for convenience:

$$P(V) = \sum_{\mathbf{X}\setminus\{\mathbf{V}\}} \phi_{\mathbf{X}}.$$

Figure 4 illustrates an example of marginalization. The cluster potential ϕ_{ABD} is from the consistent join tree of Figure 1. ϕ_{ABD} is marginalized once to compute $P(A)$, and then marginalized again to compute $P(D)$.

[12]The parent cluster of $\mathbf{F_V}$ is a convenient choice, but see Section 10.2 for a discussion of more optimal choices.

6. HANDLING EVIDENCE

We are now able to compute $P(V)$ for any variable V. In the following sections, we show how to modify the procedures in Section 5 in order to compute $P(V \mid \mathbf{e})$ in the context of evidence $\mathbf{e}$. First we introduce observations, the simplest notion of evidence, in Section 6.1. Then, in Sections 6.2–6.6, we show how to compute $P(V \mid \mathbf{e})$ for sets of observations $\mathbf{e}$. Finally, in Section 6.7, we extend the above procedures to handle more general notions of evidence.

6.1. Observations And Likelihoods

Observations are the simplest forms of evidence. An **observation** is a statement of the form $V = v$. Collections of observations may be denoted by $\mathbf{E} = \mathbf{e}$, where $\mathbf{e}$ is the instantiation of the set of variables $\mathbf{E}$. Observations are also referred to as **hard evidence**.

To encode observations in a form suitable for PPTC, we define the notion of a likelihood. Given a variable V, the **likelihood** of V, denoted as Λ_V, is a potential over $\{V\}$; in other words, Λ_V maps each value v to a real number (see Section 2.2.1). We encode an arbitrary set of observations $\mathbf{e}$ by using a likelihood Λ_V for each variable V, as follows:

- If $V \in \mathbf{E}$—that is, if V is observed—then assign each $\Lambda_V(v)$ as follows:

$$\Lambda_V(v) = \begin{cases} 1, & \text{when } v \text{ is the observed value of } V \\ 0, & \text{otherwise} \end{cases}$$

- If $V \notin \mathbf{E}$—that is, if the value of V is unknown—then assign $\Lambda_V(v) = 1$ for each value v.

Note that when there are no observations, the likelihood of each variable consists of all 1's. Table 1 illustrates how likelihoods are used to encode the observations $C = on$ and $E = \mathit{off}$, where C and E are variables from the join tree in Figure 1.

6.2. PPTC Inference With Observations

Figure 1 illustrates the overall control for PPTC with observations. We modify the control from Figure 1 to incorporate observations, as follows:

- *Initialization* (Section 6.3). We modify initialization from Section 5.2 by introducing an additional step: for each variable V, we initialize the likelihood Λ_V.

Variable V	$\Lambda_V(v)$	
	$v = on$	$v = off$
A	1	1
B	1	1
C	1	0
D	1	1
E	0	1
F	1	1
G	1	1
H	1	1

Table 1. Likelihood encoding of $C = on, E = off$.

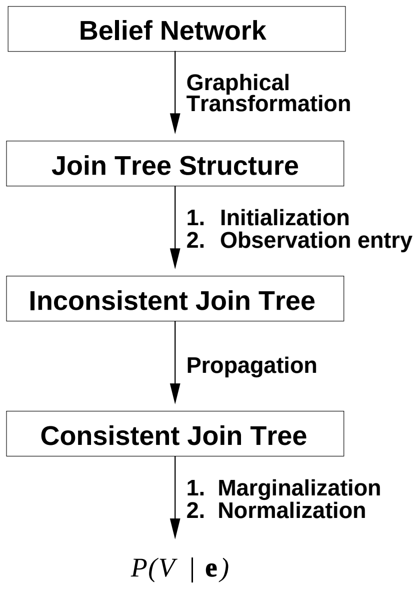


Figure 1. Block diagram of PPTC with observations.

- *Observation Entry* (Section 6.4). Following initialization, we encode and incorporate observations into the join tree; this step results in further modification of the join tree potentials.
- *Normalization* (Section 6.5). To compute $P(V \mid \mathbf{e})$ for a variable of interest V, we perform marginalization and an additional step called **normalization**.

6.3. Initialization With Observations

We keep track of observations by maintaining a likelihood for each variable. We initialize these likelihoods by adding step 2b to the initialization procedure below:

1. For each cluster and sepset $\mathbf{X}$, set each $\phi_{\mathbf{X}}(\mathbf{x})$ to 1:

$$\phi_{\mathbf{X}} \longleftarrow 1.$$

2. For each variable V:

(a) Assign to V a cluster $\mathbf{X}$ that contains $\mathbf{F_V}$; multiply $\phi_{\mathbf{X}}$ by $P(V \mid \mathbf{\Pi}_V)$:

$$\phi_{\mathbf{X}} \longleftarrow \phi_{\mathbf{X}} P(V \mid \mathbf{\Pi}_V).$$

(b) Set each likelihood element $\Lambda_V(v)$ to 1:

$$\Lambda_V \longleftarrow 1.$$

6.4. Observation Entry

Note that upon completion of initialization, the likelihoods encode no observations. We incorporate each observation $V = v$ by encoding the observation as a likelihood, and then incorporating this likelihood into the join tree, as follows:

1. Encode the observation $V = v$ as a likelihood Λ_V^{new}.

2. Identify a cluster $\mathbf{X}$ that contains V.[13]

3. Update $\phi_{\mathbf{X}}$ and Λ_V:

$$\begin{aligned} \phi_{\mathbf{X}} &\longleftarrow \phi_{\mathbf{X}} \Lambda_V^{new}. \qquad (1) \\ \Lambda_V &\longleftarrow \Lambda_V^{new}. \end{aligned}$$

[13]The parent cluster of $\mathbf{F_V}$ is a convenient choice, but see Section 10.2 for a discussion of more optimal choices.

By entering a set of observations $\mathbf{e}$ as described above, we modify the join tree potentials, so that *all subsequent probabilities derived from the join tree are probabilities of events that are conjoined with evidence* $\mathbf{e}$. In other words, instead of computing $P(\mathbf{X})$ and $P(V)$, we compute $P(\mathbf{X}, \mathbf{e})$ and $P(V, \mathbf{e})$, respectively. Note also that the join tree encodes $P(\mathbf{U}, \mathbf{e})$ instead of $P(\mathbf{U})$ (see Equation (2)).

6.5. Normalization

After the join tree is made consistent through global propagation, we have, for each cluster (or sepset) $\mathbf{X}$, $\phi_{\mathbf{X}} = P(\mathbf{X}, \mathbf{e})$, where $\mathbf{e}$ denotes the observations incorporated into the join tree according to Section 6.4 [2]. When we marginalize a cluster potential $\phi_{\mathbf{X}}$ into a variable V, we obtain the probability of V *and* $\mathbf{e}$:

$$P(V, \mathbf{e}) = \sum_{\mathbf{X} \setminus \{\mathbf{V}\}} \phi_{\mathbf{X}}.$$

Our goal is to compute $P(V \mid \mathbf{e})$, the probability of V *given* $\mathbf{e}$. We obtain $P(V \mid \mathbf{e})$ from $P(V, \mathbf{e})$ by **normalizing** $P(V, \mathbf{e})$ as follows:

$$P(V \mid \mathbf{e}) = \frac{P(V, \mathbf{e})}{P(\mathbf{e})} = \frac{P(V, \mathbf{e})}{\sum_{V} P(V, \mathbf{e})}. \qquad (2)$$

The probability of the observations $P(\mathbf{e})$ is often referred to as a **normalizing constant**.

6.6. Handling Dynamic Observations

Suppose that after computing $P(V \mid \mathbf{e_1})$, we wish to compute $P(V \mid \mathbf{e_2})$, where $\mathbf{e_2}$ is a different set of observations from $\mathbf{e_1}$. We could start anew by building a join tree structure, initializing its potentials, entering the new set of observations $\mathbf{e_2}$, performing global propagation, and marginalizing and normalizing. However, this amount of additional work is not necessary, because we can directly modify the join tree potentials in response to *changes* in the set of observations. We can imagine a dynamic system in which the consistent join tree is the steady state, and incoming observations disturb this steady state. In this subsection, we refine the control of PPTC by adding procedures to handle such dynamic observations.

6.6.1. Overall Control Figure 2 shows the control for PPTC with dynamic observations. Note that there are two dotted paths going from *consistent join tree* to *inconsistent join tree*, one labeled *global update* and the other *global retraction*. Depending on how we change the set of observations, we must perform one of these two procedures. A global update

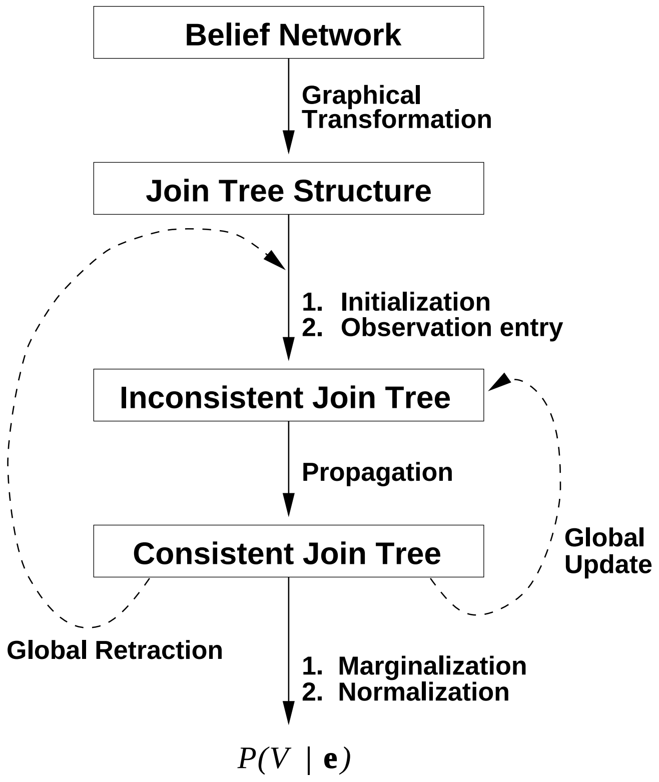


Figure 2. Block diagram of PPTC with dynamic observations.

is used to incorporate new observations, while a global retraction is required for modifying or retracting previous observations. Global retraction requires reinitialization of the join tree potentials, because undoing an observation involves restoring table elements that have been zeroed out by previous observations. To describe these procedures more precisely, we first establish some basic notions of changes in observations.

6.6.2. Updates And Retractions To describe changes in observations, we establish the notion of an observed state. The **observed state** of a variable V is its observed value v, if V is observed; otherwise, the observed state of V is unknown, and we say that V is *unobserved.*

Suppose we change a set of observations $\mathbf{e_1}$ to a different set of observations $\mathbf{e_2}$. Then the observed state of each variable V undergoes one of three changes:

- *No change.* If V is unobserved in $\mathbf{e_1}$, it remains unobserved in $\mathbf{e_2}$. If $V = v$ in $\mathbf{e_1}$, then $V = v$ in $\mathbf{e_2}$.

- *Update.* V is unobserved in $\mathbf{e_1}$, and $V = v$ in $\mathbf{e_2}$ for some value v.
- *Retraction.* $V = v_1$ in $\mathbf{e_1}$. In $\mathbf{e_2}$, V is either unobserved, or $V = v_2$, where $v_2 \neq v_1$.

We can now state how we should handle changes in observations. Suppose we have a consistent join tree that incorporates the set of observations $\mathbf{e_1}$, and we wish to compute $P(V \mid \mathbf{e_2})$ for variables of interest V, where $\mathbf{e_2}$ is different from $\mathbf{e_1}$. We incorporate $\mathbf{e_2}$ into the join tree by performing one of the following:

- *Global update* (Section 6.6.3). We perform a global update if, for each variable V, the observed state of V is unchanged or updated from $\mathbf{e_1}$ to $\mathbf{e_2}$.
- *Global retraction* (Section 6.6.4). We perform a global retraction if, for some variable V, the observed state of V is retracted.

6.6.3. Global Update A **global update** executes an observation entry (see Section 6.4 above) for each variable V whose observed state is *updated* to $V = v$. Global updating destroys the consistency of the join tree; we restore consistency by performing a global propagation. However, if the belief potential of only one cluster $\mathbf{X}$ is modified through global updating, then it is sufficient to unmark all clusters and call DISTRIBUTE-EVIDENCE($\mathbf{X}$).

6.6.4. Global Retraction We perform a **global retraction** as follows:

1. For each variable V, update the likelihood Λ_V to reflect any changes in V's observed state.

2. Reinitialize the join tree tables according to Section 6.3.

3. Incorporate each observation in $\mathbf{e_2}$ according to Section 6.4.

We cannot handle retractions in the same way that we handle updates, because in a retraction, we are trying to recover join tree potential elements that have been zeroed out by previous observations. Our only recourse, therefore, is to reinitialize the join tree tables and then enter the new set of observations.[14]

[14]Observations can be introduced without zeroing out the cluster tables by using alternative propagation methods—examples include “fast retraction” [24, 25] and “cautious propagation” [26]. Applications of these methods include conflict analysis, sensitivity analysis, and processing counterfactual queries. Compared to PPTC propagation, these alternative methods require more storage and computation, and generally do not handle all possible retractions of observations.

6.7. More Sophisticated Notions Of Evidence

Observations are the simplest and most common type of evidence encountered. However, we can use likelihoods to represent more sophisticated types of evidence. We introduce these more general notions of evidence in Sections 6.7.1. Then we describe PPTC with evidence in a manner parallel to our description of PPTC with observations.

6.7.1. Evidence And Likelihoods In general, a likelihood Λ_V on a variable V can be interpreted as assigning a relative weight $\Lambda_V(v)$ to each value v, according to how likely is the case that $V = v$. When $\Lambda_V(v)$ is the same for all values v, we say that Λ_V encodes no information on variable V.[15]

The following terminology is commonly used to classify different types of evidence on a variable V, according to the form that the likelihood Λ_V takes:

- **Virtual Evidence**. Virtual evidence, or **soft evidence**, is the most general type of evidence. Virtual evidence on a variable V is represented by a likelihood Λ_V where each $\Lambda_V(v)$ is a real number in $[0, 1]$ [4].
- **Finding**. A finding is represented by a likelihood Λ_V where each $\Lambda_V(v)$ is a 0 or a 1. Essentially, a finding declares the zeroed values to be impossible. Naturally, a finding should allow at least one possible value.
- **Observation**. An observation is a special case of a finding where $\Lambda_V(v) = 1$ for exactly one value v. An observation declares, with certainty, that $V = v$.

6.7.2. Evidence Entry Upon completion of initialization, no evidence is encoded by the likelihoods or incorporated into the join tree. We incorporate each piece of evidence on a variable V by executing the following procedure:

1. Encode the evidence on variable V as a likelihood Λ_V^{new}.

2. Identify a cluster $\mathbf{X}$ that contains V.[16]

3. Update $\phi_{\mathbf{X}}$ and Λ_V:

$$\begin{aligned} \phi_{\mathbf{X}} &\longleftarrow \phi_{\mathbf{X}} \frac{\Lambda_V^{new}}{\Lambda_V}. \\ \Lambda_V &\longleftarrow \Lambda_V^{new}. \end{aligned} \tag{3}$$

Note that Equation (1) is a specialized version of Equation (3) above.

[15]Typically, we encode this no-information state with a vector of 1's.

[16]The parent cluster of $\mathbf{F_V}$ is a convenient choice, but see Section 10.2 for a discussion of more optimal choices.

6.7.3. Handling Dynamic Evidence We can easily extend PPTC with dynamic observations (Section 6.6) to handle dynamic evidence. In discussing changes of evidence, we generalize the notion of observed state to the notion of evidence state: the **evidence state** of a variable V is its likelihood Λ_V. We extend the notation $\mathbf{e}$ to represent the combined evidence state of *all* variables, and we refer to $\mathbf{e}$ as an **evidence configuration**.

Consider a change in evidence configuration from $\mathbf{e_1}$ to $\mathbf{e_2}$. For each variable V, denote its evidence state in $\mathbf{e_1}$ as Λ_V, and its evidence state in $\mathbf{e_2}$ as Λ_V^{new}. We classify the change from Λ_V to Λ_V^{new} as one of the following:

- *No change.* $\Lambda_V = \Lambda_V^{new}$.
- *Update.* For every value v, $\Lambda_V(v) = 0$ implies $\Lambda_V^{new}(v) = 0$.
- *Retraction.* For some value v, $\Lambda_V(v) = 0$ and $\Lambda_V^{new}(v) \neq 0$.

We can now state how we should handle changes in evidence. Suppose we have a consistent join tree that incorporates the evidence configuration $\mathbf{e_1}$, and we wish to compute $P(V \mid \mathbf{e_2})$ for variables of interest V, where $\mathbf{e_2}$ is different from $\mathbf{e_1}$. We incorporate $\mathbf{e_2}$ into the join tree by performing one of the following:

- *Global update.* We perform a global update if, for each variable V, the evidence state of V is unchanged or updated from $\mathbf{e_1}$ to $\mathbf{e_2}$.
- *Global retraction.* We perform a global retraction if, for some variable V, the evidence state of V is retracted.

We perform a global update by executing an *evidence entry* (Section 6.7.2) for each variable V whose evidence state is updated from Λ_V to Λ_V^{new}. We perform a global retraction as follows:

1. For each variable V, update the likelihood Λ_V to reflect any changes in V's observed state.

2. Reinitialize the join tree tables according to Section 6.3.

3. For each variable V where $\Lambda_V(v) \neq 1$ for some v, incorporate Λ_V according to Section 6.7.2.

7. PPTC OPTIMIZATIONS

In this section, we discuss some optimizations to PPTC that we implemented, optimizations that can significantly reduce the computation required for inference in certain situations. We assume that the reader has mastered the material in the previous sections and has a basic understanding of computer algorithms.

7.1. Query-Driven Message Passing

In this section, we summarize a modified version of PPTC called **query-driven PPTC**. Unlike the version of PPTC presented in previous sections, query-driven PPTC does not establish and maintain consistency throughout the join tree; instead, it passes messages only in response to individual **variable queries** $P(V \mid \mathbf{e})$. This optimization is useful in diagnostic applications: for example, where the user constructs a belief network with many variables, and then queries only a few variables.

Query-driven PPTC exploits the following observation: to marginalize the cluster potential $\phi_{\mathbf{X}}$ to obtain $P(V, \mathbf{e})$, we need only to ensure that $\phi_{\mathbf{X}} = P(\mathbf{X}, \mathbf{e})$. A call to Collect-Evidence($\mathbf{X}$) would ensure this condition [23]. However, query-driven PPTC uses a modified version of Collect-Evidence($\mathbf{X}$) that recurses on a neighbor $\mathbf{Y}$ only if $\mathbf{Y}$ has not previously passed a message to $\mathbf{X}$. Query-driven PPTC keeps track of the messages that have been passed by maintaining a set of Boolean quantities called **message flags**. Each message flag is denoted as $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$ and is interpreted as follows: the message flag $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$ is $TRUE$ if a message pass from $\mathbf{Y}$ to $\mathbf{X}$ would leave $\phi_{\mathbf{X}}$ unchanged; otherwise, $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$ is $FALSE$. We use the notation $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$ to emphasize that the message flags can be stored locally: given a cluster $\mathbf{X}$, we can store the message flags $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$, for all neighbors $\mathbf{Y}$, as part of the local information on $\mathbf{X}$.

A message flag $\mathcal{M}_{\mathbf{X}}(\mathbf{Y})$ is set to $TRUE$ during a message pass from $\mathbf{X}$ to $\mathbf{Y}$. As additional variable queries are processed, additional message flags are set to $TRUE$. Message flags, however, can be set to $FALSE$, or **invalidated**, by dynamic evidence:

- Evidence update. Suppose a cluster $\mathbf{X}$ incorporates an evidence update according to the procedure in Section 6.7.2. Then all message passes in the direction *away* from cluster $\mathbf{X}$ are invalidated; these message passes need to be recomputed if a subsequent variable query requests them.

- Evidence retraction. To process changes in evidence that involve retraction, we employ the familiar procedure of reinitializing the join tree tables and entering the evidence anew. All message passes are invalidated.

7.2. Inference On Forests Of Join Trees

If the initial belief network is not fully connected, then the procedures in Section 4 yield a join tree with empty sepsets. We can optimize PPTC by disallowing these empty sepsets and performing inference on a *forest* of join trees. By maintaining each join tree separately, we avoid the computational cost of passing messages that serve only to rescale the cluster potentials.

In maintaining separate join trees, we must also, in general, maintain separate normalization constants for each join tree [27]. First, we note that the normalization constant $P(\mathbf{e})$ for a join tree that incorporates evidence $\mathbf{e}$ can be computed, using any cluster $\mathbf{X}$ where $\phi_{\mathbf{X}} = P(\mathbf{X}, \mathbf{e})$, as follows:

$$P(\mathbf{e}) = \sum_{\mathbf{X}} P(\mathbf{X}, \mathbf{e}) = \sum_{\mathbf{X}} \phi_{\mathbf{X}}.$$

Therefore, $P(\mathbf{e})$ can be computed by calling COLLECT-EVIDENCE on some cluster $\mathbf{X}$, and then marginalizing $\phi_{\mathbf{X}}$ as described above. But this marginalization effectively occurs during the normalization phase of a variable query, as seen in the denominator of Equation (2).

Now let's consider a forest of join trees $\mathcal{T}_1, \ldots, \mathcal{T}_n$ with corresponding normalization constants $P(\mathbf{e_1}), \ldots, P(\mathbf{e_n})$. Since the disconnected join trees are independent of one another, the probability of evidence $P(\mathbf{e})$ for the entire join forest can be calculated as follows:

$$P(\mathbf{e}) = P(\mathbf{e_1} \ldots \mathbf{e_n}) = \prod_{i=1}^{n} P(\mathbf{e_i}).$$

Suppose we query a variable V in $\mathcal{T}_1$. We choose a cluster $\mathbf{X}$ that contains V, call COLLECT-EVIDENCE($\mathbf{X}$), and obtain $\phi_{\mathbf{X}} = P(V, \mathbf{e_1})$. But if we want to compute $P(V, \mathbf{e})$, we must also compute the other normalization constants:

$$\begin{aligned} P(V, \mathbf{e}) &= P(V, \mathbf{e_1} \ldots \mathbf{e_n}) = P(V, \mathbf{e_1} \mid \mathbf{e_2} \ldots \mathbf{e_n}) P(\mathbf{e_2} \ldots \mathbf{e_n}) \\ &= P(V, \mathbf{e_1}) P(\mathbf{e_2} \ldots \mathbf{e_n}) = P(V, \mathbf{e_1}) \prod_{i \neq 1} P(\mathbf{e_i}), \end{aligned}$$

where each normalization constant $P(\mathbf{e_i})$ is computed by marginalizing a cluster in the join tree $\mathcal{T}_i$.

However, if we are interested only in computing $P(V \mid \mathbf{e})$, we do *not* need the other normalization constants:

$$P(V \mid \mathbf{e}) = \frac{P(V, \mathbf{e})}{P(\mathbf{e})} = \frac{P(V, \mathbf{e_1}) P(\mathbf{e_2} \ldots \mathbf{e_n})}{P(\mathbf{e_1} \mid \mathbf{e_2} \ldots \mathbf{e_n})} = \frac{P(V, \mathbf{e_1})}{P(\mathbf{e_1})}.$$

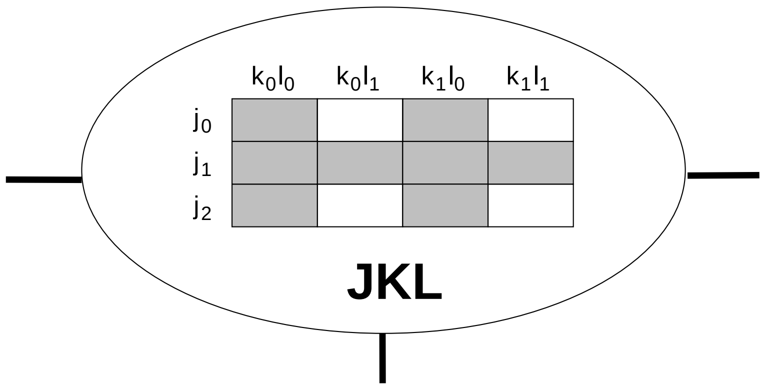


Figure 1. Evidence shrinking on a single cluster. Cluster JKL is shrunk by the finding $J \neq j_1$ and the observation $L = l_1$, reducing its effective size from 12 to 4. The unshaded cells represent the cluster elements that remain active after the shrinking process.

7.3. Evidence Shrinking

Evidence shrinking is an optimization of PPTC that uses findings (and observations) to reduce effective cluster *sizes.* As an example, let us focus on a particular cluster JKL in a join tree that has just processed the findings $J \neq j_1$ and $L = l_1$ (Figure 1). How should this evidence affect cluster JKL? Mathematically, we would multiply all of the shaded cells in Figure 1 by zero. But in practice, we do not want to do this, because the 0's will not affect the *results* of subsequent message passes involving JKL. Both the introduction of these 0's and their subsequent propagation would involve unnecessary, and often, costly computation.

Evidence shrinking avoids this unnecessary processing of 0s by restructuring the cluster JKL so that only the *unshaded* cells—the cells that would *not* have been multipled by 0—are involved in further computation. This restructuring process can be performed in time proportional to the *reduced* cluster size. Further details on this restructuring are discussed in Section 8.2.

Two properties of evidence shrinking contribute to its potential for significant computational savings. First, the 0's in a likelihood Λ_V affects *all* clusters containing the variable V. Second, if we restrict our evidence to observations, as is the case for many existing implementations, then each observation on a variable V effectively reduces the size of each cluster (and sepset) containing V by an entire *dimension.* These two properties of evidence shrinking are illustrated in Figure 2.

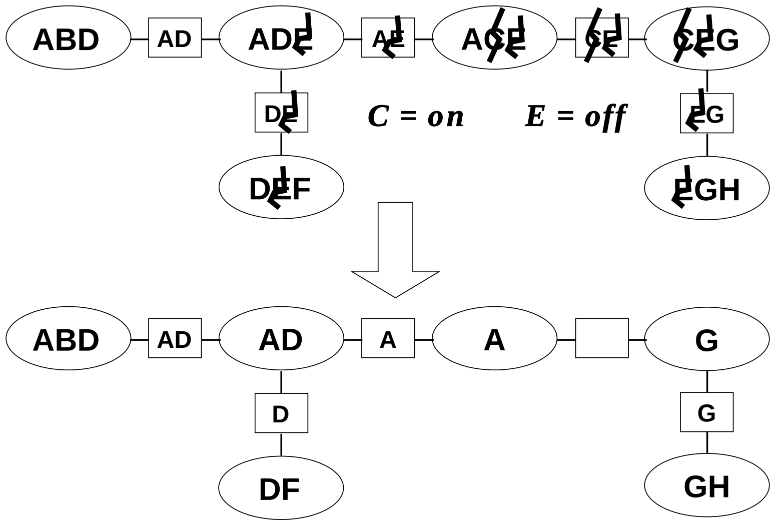


Figure 2. Evidence shrinking on a join tree. Observation $C = on$ reduces the matrices of clusters containing C by a dimension. Similarly, observation $E = off$ reduces the matrices of clusters containing E by a dimension. Note that the sepset CE becomes empty; it passes a message composed of a single number $P(C = on, E = off \mid e_{T_1})$, where e_{T_1} is the evidence in the subtree T_1 from which the message through CE originates.

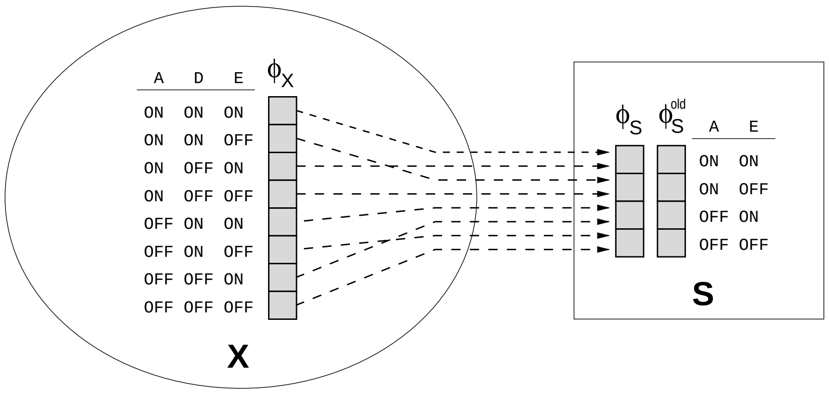


Figure 1. Interaction between cluster and sepset arrays. **X** denotes a cluster ADE and **S** denotes a neighboring sepset AE. The shaded cells denote potential elements; next to each potential element is the instantiation of variables that indexes it. Each cluster element $\phi_{\mathbf{X}}(\mathbf{x})$ has a corresponding sepset element $\phi_{\mathbf{S}}(\mathbf{s})$ (and $\phi_{\mathbf{S}}^{old}(\mathbf{s})$), where $\mathbf{s}$ is consistent with $\mathbf{x}$; this correspondence is illustrated by the dashed arrows.

8. ARRAY-LEVEL TECHNIQUES

"The devil is in the details," it is often said. This is definitely the case when implementing PPTC. In this section, we address some array-level issues that are not normally discussed in the probabilistic literature; yet, they must be addressed by any programmer who wishes to build an efficient implementation of PPTC. We present some techniques that, through additional precomputation prior to inference, can reduce the overhead during message passing. Additional array-level techniques are presented in Section 10.1.

8.1. Cluster-Sepset Mappings

In this section we describe an auxiliary data structure, called a **cluster-sepset** mapping, that is crucial to an efficient implementation of PPTC inference. Recall that a message pass consists of two steps: projection and absorption (see Section 5.3.1). Both projection and absorption depend on a precise interaction between a cluster potential and a sepset potential. These potentials are typically implemented as arrays, and the interaction between these arrays is illustrated in Figure 1.

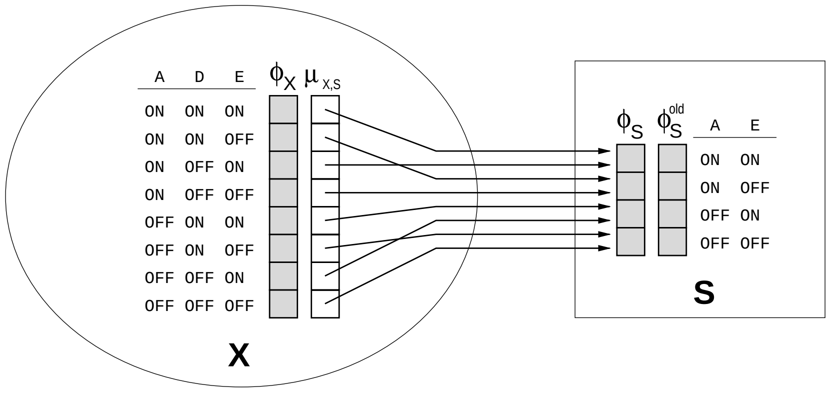


Figure 2. Example of a cluster-sepset mapping. For each cluster instantiation $\mathbf{x}$, the cluster-sepset mapping $\mu_{\mathbf{X},\mathbf{S}}$ stores the array index of the consistent sepset instantiation $\mathbf{s}$. The solid arrows illustrate the resulting correspondence between between cluster elements $\phi_{\mathbf{X}}(\mathbf{x})$ and sepset elements $\phi_{\mathbf{S}}(\mathbf{s})$. Both projection and absorption use the same cluster-sepset mapping; the only difference is in the actual arithmetic performed.

In both projection and absorption, the key step is to locate, for each cluster element $\phi_{\mathbf{X}}(\mathbf{x})$, the corresponding sepset element $\phi_{\mathbf{S}}(\mathbf{s})$ (and $\phi_{\mathbf{S}}^{old}(\mathbf{s})$), where $\mathbf{s}$ is consistent with $\mathbf{x}$. But in order to locate $\phi_{\mathbf{S}}(\mathbf{s})$, we need not just the instantiation $\mathbf{s}$, but the *array index* of $\mathbf{s}$. Computing the array index of $\mathbf{s}$ requires a number of operations involving $\mathbf{x}$ and the array dimensions of $\mathbf{X}$ and $\mathbf{S}$. For a given message pass, this computation needs to be applied to each cluster instantiation $\mathbf{x}$. Furthermore, these array indices must be recomputed during the next message pass involving $\mathbf{X}$ and $\mathbf{S}$, unless they are somehow stored for future retrieval.

We avoid unnecessary recomputation of these array indices by *precomputing* them while building the join tree. Specifically, for each cluster $\mathbf{X}$ and neighboring sepset $\mathbf{S}$, we compute an array $\mu_{\mathbf{X},\mathbf{S}}$ over the instantiations $\mathbf{x}$, such that each array element $\mu_{\mathbf{X},\mathbf{S}}(\mathbf{x})$ stores the array index of the instantiation $\mathbf{s}$ that is consistent with $\mathbf{x}$. We call $\mu_{\mathbf{X},\mathbf{S}}$ a **cluster-sepset mapping**. Figure 2 illustrates an example of a cluster-sepset mapping.

A cluster-sepset mapping $\mu_{\mathbf{X},\mathbf{S}}$ can be computed in time proportional to the number of instantiations of $\mathbf{X}$. Cluster-sepset mappings significantly reduce the running time of inference because they enable corresponding elements to be located using simple array lookups, not repeated array index calculations.

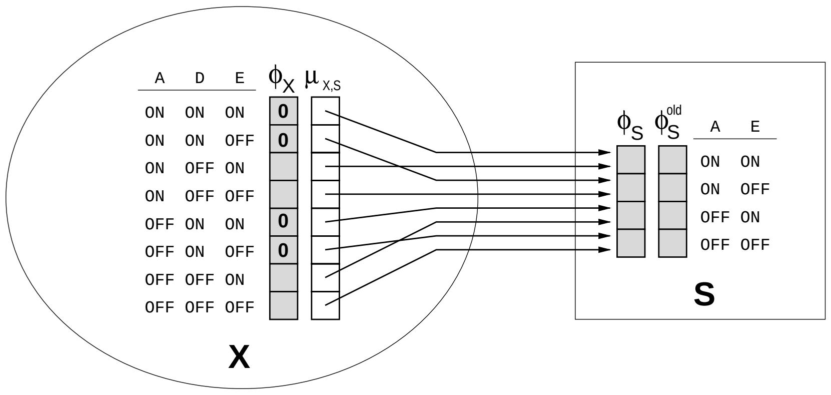


Figure 3. Zeroing of cluster elements by the observation $D = \mathit{off}$.

8.2. Evidence Shrinking

Suppose we incorporate the observation $D = \mathit{off}$ into the cluster **X** from Figure 2, using the observation entry procedure in Section 6.4. This would cause certain elements of **X** to be zeroed, as illustrated in Figure 3. These 0's will continue to be visited during subsequent message passes involving **X**, even though they will not affect the results of any computations.

Evidence shrinking (Section 7.3) seeks to avoid these extraneous and costly element accesses. The computational gains of evidence shrinking hinge on restructuring the clusters so that only the **active elements**—the elements that are not zeroed by the evidence—are visited during subsequent message passes. We can implement this restructuring by maintaining, for each cluster, an auxiliary array of indices called a **shrink mapping**. A shrink mapping on a cluster **X** is an array $\sigma_{\mathbf{X}}$ that points to the active elements of $\phi_{\mathbf{X}}$. The effective size of the shrink mapping is the number of active elements in **X**. During projection or absorption, the active elements of **X** are accessed by visiting the elements of the shrink mapping. The shrink mapping amounts to an additional level of indirection. Figure 4 illustrates an example of a shrink mapping.

Shrink mappings can be updated in time proportional to the *reduced* cluster size. One programming solution involves using a procedure that generates all instantiations of the cluster variables and their element indices by recursing over the values of each variable. We would implement evidence shrinking by modifying this procedure to recurse only over the *possible* values of the variables in that cluster.

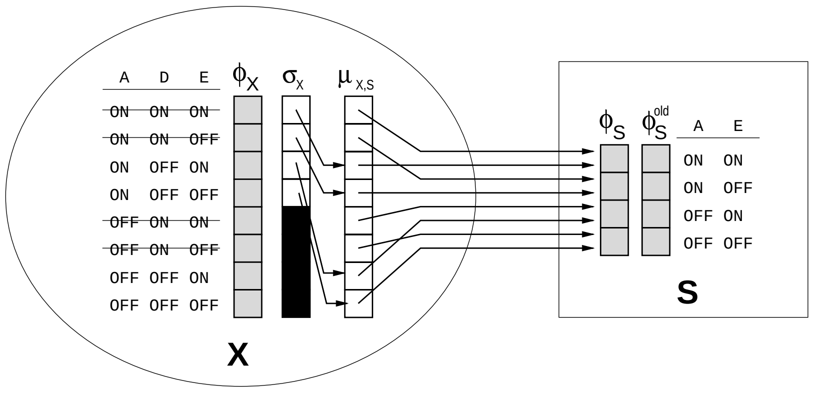


Figure 4. Shrink mapping example. Cluster $\mathbf{X}$ is shrunk by the observation $D = \mathit{off}$. The shrink mapping $\sigma_{\mathbf{X}}$ points to the cluster elements consistent with this observation. This reduces the effective cluster size, and with it, the running time of any message passes involving that cluster.

9. CONCLUSION

PPTC is one of the most recognized algorithms for exact probabilistic inference in belief networks. In this document, we provided a self-contained, procedural guide to understanding and implementing PPTC. We synthesized methods that are scattered throughout the literature, and we articulated these methods in algorithmic form. In addition, we discussed undocumented, lower-level issues that are vital to producing a robust and efficient implementation of PPTC. We hope that this document makes probabilistic inference more accessible to the increasing number of researchers and developers who are making use of this technology.

ACKNOWLEDGEMENTS

Cecil Huang thanks:

- Adnan Darwiche, for his inspiration and intimate involvement in producing this document.
- Finn Jensen, Frank Jensen, Mark Peot, and Ross Shachter, for sharing their expertise in probabilistic inference.
- Adam Galper, Paul Dagum, Sampath Srinivas, and Ted Shortliffe, for helpful discussions.
- Bruce Seely, Moisés Goldszmidt, and Serdar Uckun, for feedback on earlier versions of this document.
- the anonymous reviewers, for their contributions to the final version of this document.

This work was funded by the Medical Scientist Training Program grant #5T32GM07365 from the National Institute of General Medical Sciences, and from Rockwell Science Center in Palo Alto and Thousand Oaks, California. Computing facilities were provided by Rockwell Science Center and the CAMIS resource under NIH grant LM-05305.

10. ADDITIONAL OPTIMIZATIONS

In this appendix, we outline some additional optimization opportunities for PPTC and provide the relevant references.

10.1. Zero compression

Zero compression is an extension of PPTC that can save significant computation under certain circumstances. Here, we summarize the basic ideas of zero compression, but the interested reader can find a more thorough treatment in the original paper by Jensen and Andersen [28].

Zero compression is designed to take advantage of conditional probability tables $P(V \mid \mathbf{\Pi}_V)$ whose row entries:

- contain 0's, implying some logical or functional relationship between variable V and its parents $\mathbf{\Pi}_V$;
- contain extreme probabilities that are close to 0.

These situations occur often in practice; for example, in engineering applications that model small failure probabilities.

10.1.1. Annihilating zeros During initialization, each conditional probability $P(V \mid \mathbf{\Pi}_V)$ is multipled into some cluster potential $\phi_{\mathbf{X}}$. Let us first focus on a particular conditional probability distribution $P(V \mid \mathbf{\Pi}_V)$. The 0's in $P(V \mid \mathbf{\Pi}_V)$ cause the corresponding elements in $\phi_{\mathbf{X}}$ to be zero as well. After performing global propagation, some of these 0's will propagate throughout the entire join tree.

Suppose now, that the user enters evidence and performs another global propagation. During this propagation, computational resources are expended adding and multiplying potential elements by 0. This expenditure becomes more wasteful as the number of 0s increases. Zero compression, as presented in [28], addresses this wasteful propagation as follows:

1. Build a join tree. Initialize the cluster potentials with the conditional probabilities $P(V \mid \mathbf{\Pi}_V)$.

2. Perform a global propagation—a Collect-Evidence followed by a Distribute-Evidence.

3. For each cluster $\mathbf{X}$, visit each element $\phi_{\mathbf{X}}(\mathbf{x})$, identifying and annihilating the 0 elements. The annihilation step should restructure the internals of $\mathbf{X}$ so that subsequent messages passes involving $\mathbf{X}$ do *not* visit these 0 elements.

10.1.2. Annihilating "small" elements Zero compression can speed up exact inference in a join tree because its effective cluster sizes are reduced. We can reduce the effective cluster sizes further by annihilating elements with probabilities *close* to zero; this elimination of "small" elements results in a join tree that performs *approximate* inference. Details on how to select appropriate annihilation thresholds for each cluster are contained in [28]. Note that unlike the strict zero-compression scheme in Section 10.1.1 above, annihiliating small elements destroys the consistency of the join tree. This loss of consistency can be remedied by a global propagation; in the case of query-driven PPTC, this loss of consistency can be properly accounted for by invalidating the appropriate message flags.

The above approximation scheme can result in significant computational gains, depending on the topology and quantification of the original belief network and the amount of error tolerated by the user. In some scenarios, the total number of elements *not* annihilated may be orders of magnitude smaller than the original number of elements. For example, [28] discusses some experiments on a real-world belief network, in which the inference time is reduced by 96–99 percent for a total removed probability mass of 0.1 percent.

10.1.3. Dynamic zero compression With appropriate data structuring, a form of zero compression that dynamically compresses cluster matrices during *inference* can be implemented. When a cluster element evaluates to 0 or a sufficiently small number, that element would be annihilated immediately.

10.2. Dynamic restructuring of cluster trees

Recall that for marginalization and evidence entry, we are asked to "choose a cluster $\mathbf{X}$ that contains the variable V." In each of these situations, we conveniently chose the parent cluster of V. However, by choosing these clusters more judiciously, we can often, for a given query, reduce the number of message passes, or choose message passes involving smaller clusters.[17] The range of message-passing options expands further if we allow the possibility of dynamically restructuring cluster trees by translocating sepsets in a manner that preserves the join tree property [27].

10.3. Optimizations at the arithmetic-expression level

The join tree is a convenient, intermediate structure for performing inference on multiply-connected belief networks. Its construction is validated by fundamental results from the theory of conditional independence [12], and the local message-passing and marginalization strategies are both intuitive and mathematically well-founded. However, this formulation of the inference problem often masks additional opportunities for optimization. D'Ambrosio exposes some of these opportunities by redefining the inference task at a "smaller grain size": optimizing the computation of individual terms, as opposed to individual marginal distributions [29]. Given this formulation, the challenge is to construct optimal arithmetic expressions for specific queries, taking advantage of conditional independencies and partial results cached from previous computations. Li and D'Ambrosio present one approach in their recent improvement of the SPI algorithm [10]. Darwiche and Provan also address probabilistic inference at the arithmetic-expression level [30]; their approach generates and optimizes expression dags off-line, then evaluates these dags on-line in response to dynamic evidence. They describe a method, based on PPTC, for generating such expressions; these expressions can be used to answer queries with respect to evidence about a predefined set of variables. The size of a generated expression, using their method, is proportional to the total size of the cluster and sepset tables in the join tree. More importantly, the method they use for updating these expressions associates validity flags with individual arithmetic operations, thus leading to optimizations that are more refined than those achieved by the message flags, as suggested in Section 7.1.

[17]Note that variables can be viewed as sepsets: a cluster incorporates evidence on V by "absorbing" from V, and a probability distribution $P(V)$ is computed by "projecting" from an appropriate cluster.

References

1. Lauritzen, S. L., and Spiegelhalter, D. J., Local computations with probabilities on graphical structures and their application to expert systems, *J. Roy. Stat. Soc. B*, 50, 157–224, 1988.

2. Jensen, F. V., Lauritzen, S. L., and Olesen, K. G., Bayesian updating in causal probabilistic networks by local computations, *Comp. Stat. Quart.*, 4, 269–282, 1990.

3. Shenoy, P., and Shafer, G., Axioms for probability and belief-function propagation, in *Uncertainty and Artificial Intelligence, Vol. 4* (R. D. Shachter, T. Levitt, J. F. Lemmer and L. N. Kanal, Eds.), Elsevier, North-Holland, Amsterdam, 169–198, 1990.

4. Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Second edition, Morgan Kaufmann, San Mateo, Calif., 1988.

5. Neapolitan, R. E., *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*, John Wiley and Sons, New York, 1990.

6. Pearl, J., A constraint-propagation approach to probabilistic reasoning, in *Uncertainty and Artificial Intelligence* (L. N. Kanal and J. F. Lemmer, Eds.), Elsevier, New York, 357–369, 1986.

7. Peot, M. A., and Shachter, R. D., Fusion and propagation with multiple observations in belief networks, *Artif. Intell.*, 48(3), 299–318, 1991.

8. Darwiche, A., Conditioning algorithms for exact and approximate inference, in *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, 99–107, 1995.

9. Shachter, R., D'Ambrosio, B., and Del Favero, B., Symbolic probabilistic inference in belief networks, in *Proceedings of the 8th National Conference on AI*, Boston, Mass., 126–131, 1990.

10. Li, Z., and D'Ambrosio, B., Efficient inference in bayes nets as a combinatorial optimization problem, *Int. J. Approximate Reasoning*, 11(1), 55–81, 1994.

11. Dagum, P., and Horvitz, E., A bayesian analysis of simulation algorithms for inference in belief networks, *Networks*, 23, 499–516, 1993.

12. Dawid, A. P., Conditional independence in statistical theory, *J. Roy. Stat. Soc. B*, 41(1), 1–33, 1979.

13. Geiger, D., Verma, T., and Pearl, J., Identifying independence in bayesian networks, *Networks*, 20, 507–534, 1990.

14. Charniak, E., Bayesian networks without tears, *AI Mag.*, 12(4), 50–63, 1991.

15. Shachter, R. D., Background review and terminology, in *Making Decisions in Intelligent Systems: Representing Uncertainty with Belief Networks and Influence Diagrams*, Duxbury, Belmont, Calif. Manuscript in progress.

16. Lauritzen, S. L., Dawid, A. P., Larsen, B. N., and Leimer, H.-G., Independence properties of directed markov fields, *Networks*, 20, 491–505, 1990.

17. Kjærulff, U., Triangulation of graphs—algorithms giving small total state space, Technical Report R-90-09, Dept. of Math. and Comp. Sci., Aalborg University, Denmark, 1990.

18. Cormen, T. H., Leiserson, C. E., and Rivest, R. L., Heapsort, in *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 140–152, 1990.

19. Arnborg, S., Corneil, D. G., and Proskurowski, A., Complexity of finding embeddings in a k-tree, *SIAM Journal of Algebraic and Discrete Methods*, 8(2), 277–284, 1987.

20. Golumbic, M. C., Triangulated graphs, in *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 98–100, 1980.

21. Tarjan, R. E., and Yannakakis, M., Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce hypergraphs, *SIAM J. Comp.*, 13(3), 566–579, 1984.

22. Jensen, F. V., and Jensen, F., Optimal junction trees, in *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, Seattle, Wash., 360–366, 1994.

23. Jensen, F. V., Propagation in DAGs, in *Introduction to Bayesian Networks*, UCL Press, London. Manuscript in progress.

24. Cowell, R. G., and Dawid, A. P., Fast retraction of evidence in a probabilistic expert system, *Statistics and Computing*, 2, 37–40, 1992.

25. Jensen, F., Implementation aspects of various propagation algorithms in hugin, Technical Report R-94-2014, Dept. of Math. and Comp. Sci., Aalborg University, Denmark, 1994.

26. Jensen, F. V., Cautious propagation in bayesian networks, in *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, Montreal, Canada, 323–328, 1995.

27. Shachter, R. D., Andersen, S. K., and Szolovits, P., Global conditioning for probabilistic inference in belief networks, in *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann, Seattle, Wash., 514–521, 1994.

28. Jensen, F., and Andersen, S. K., Approximations in bayesian belief universes for knowledge-based systems, in *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence*, Cambridge, Mass., 162–169, 1990.

29. D'Ambrosio, B., Incremental probabilistic inference, in *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence*, 301–308, 1993.

30. Darwiche, A., and Provan, G., Query dags: A practical paradigm for implementing belief-network inference, in *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence*, 1996. to appear.