10 *Exact Inference: Clique Trees*

In the previous chapter, we showed how we can exploit the structure of a graphical model to perform exact inference effectively. The fundamental insight in this process is that the factorization of the distribution allows us to perform local operations on the factors defining the distribution, rather than simply generate the entire joint distribution. We implemented this insight in the context of the *variable elimination* algorithm, which sums out variables one at a time, multiplying the factors necessary for that operation.

In this chapter, we present an alternative implementation of the same insight. As in the case of variable elimination, the algorithm uses manipulation of factors as its basic computational step. However, the algorithm uses a more global data structure for scheduling these operations, with surprising computational benefits.

Throughout this chapter, we will assume that we are dealing with a set of factors Φ over a set of variables $\mathcal{X}$, where each factor ϕ_i has a scope $\boldsymbol{X}_i$. This set of factors defines a (usually) unnormalized measure

$$\tilde{P}_\Phi(\mathcal{X}) = \prod_{\phi_i \in \Phi} \phi_i(\boldsymbol{X}_i). \tag{10.1}$$

For a Bayesian network without evidence, the factors are simply the CPDs, and the measure $\tilde{P}_\Phi$ is a normalized distribution. For a Bayesian network $\mathcal{B}$ with evidence $\boldsymbol{E} = \boldsymbol{e}$, the factors are the CPDs restricted to $\boldsymbol{e}$, and $\tilde{P}_\Phi(\mathcal{X}) = P_\mathcal{B}(\mathcal{X}, \boldsymbol{e})$. For a Gibbs distribution (with or without evidence), the factors are the (restricted) potentials, and $\tilde{P}_\Phi$ is the unnormalized Gibbs measure.

It is important to note that all of the operations that one can perform on a normalized distribution can also be performed on an unnormalized measure. In particular, we can marginalize $\tilde{P}_\Phi$ on a subset of the variables by summing out the others. We can also consider a conditional measure, $\tilde{P}_\Phi(\boldsymbol{X} \mid \boldsymbol{Y}) = \tilde{P}_\Phi(\boldsymbol{X}, \boldsymbol{Y})/\tilde{P}_\Phi(\boldsymbol{Y})$ (which, in fact, is the same as $P_\Phi(\boldsymbol{X} \mid \boldsymbol{Y})$).

10.1 Variable Elimination and Clique Trees

Recall that the basic operation of the variable elimination algorithm is the manipulation of factors. Each step in the computation creates a factor ψ_i by multiplying existing factors. A variable is then eliminated in ψ_i to generate a new factor τ_i, which is then used to create another factor. In this section, we present another view of this computation. We consider a factor ψ_i to be a computational data structure, which takes *"messages"* τ_j generated by other factors ψ_j, and generates a message τ_i that is used by another factor ψ_l.

message

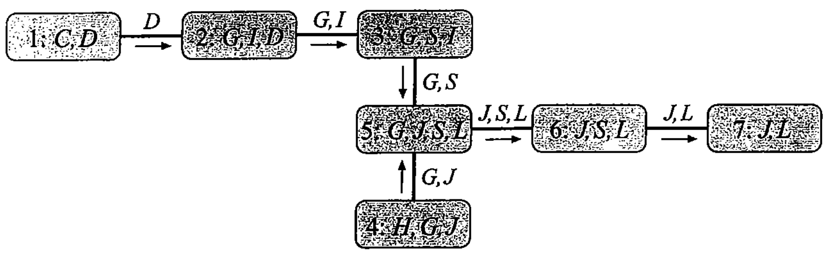


Figure 10.1 Cluster tree for the VE execution in table 9.1

10.1.1 Cluster Graphs

We begin by defining a *cluster graph* — a data structure that provides a graphical flowchart of the factor-manipulation process. Each node in the cluster graph is a *cluster*, which is associated with a subset of variables; the graph contains undirected edges that connect clusters whose scopes have some non-empty intersection. We note that this definition is more general than the data structures we use in this chapter, but this generality will be important in the next chapter, where we significantly extend the algorithms of this chapter.

Definition 10.1
cluster graph
family preservation
sepset

A cluster graph $\mathcal{U}$ *for a set of factors* Φ *over* $\mathcal{X}$ *is an undirected graph, each of whose nodes* i *is associated with a subset* $\boldsymbol{C}_i \subseteq \mathcal{X}$. *A cluster graph must be* family-preserving — *each factor* $\phi \in \Phi$ *must be associated with a cluster* $\boldsymbol{C}$, *denoted* $\alpha(\phi)$, *such that Scope*$[\phi] \subseteq \boldsymbol{C}_i$. *Each edge between a pair of clusters* $\boldsymbol{C}_i$ *and* $\boldsymbol{C}_j$ *is associated with a* sepset $\boldsymbol{S}_{i,j} \subseteq \boldsymbol{C}_i \cap \boldsymbol{C}_j$. ■

An execution of variable elimination defines a cluster graph: We have a cluster for each factor ψ_i used in the computation, which is associated with the set of variables $\boldsymbol{C}_i = Scope[\psi_i]$. We draw an edge between two clusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ if the message τ_i, produced by eliminating a variable in ψ_i, is used in the computation of τ_j.

Example 10.1

Consider the elimination process of table 9.1. In this case, we have seven factors $\psi_1, \ldots, \psi_7$, *whose scope is shown in the table. The message* $\tau_1(D)$, *generated from* $\psi_1(C, D)$, *participates in the computation of* ψ_2. *Thus, we would have an edge from* $\boldsymbol{C}_1$ *to* $\boldsymbol{C}_2$ *Similarly, the factor* $\tau_3(G, S)$ *is generated from* ψ_3 *and used in the computation of* ψ_5. *Hence, we introduce an edge between* $\boldsymbol{C}_3$ *and* $\boldsymbol{C}_5$. *The entire graph is shown in figure 10.1. The edges in the graph are annotated with directions, indicating the flow of messages between clusters in the execution of the variable elimination algorithm. Each of the factors in the initial set of factors* Φ *is also associated with a cluster* $\boldsymbol{C}_i$. *For example, the cluster* $\phi_D(D, C)$ *(corresponding to the CPD* $P(D \mid C)$*) is associated with* $\boldsymbol{C}_1$, *and the cluster* $\phi_H(H, G, J)$ *(corresponding to the CPD* $P(H \mid G, J)$*) is associated with* $\boldsymbol{C}_4$. ■

10.1.2 Clique Trees

The cluster graph associated with an execution of variable elimination is guaranteed to have certain properties that turn out to be very important.

First, recall that the variable elimination algorithm uses each intermediate factor τ_i at most once: when ϕ_i is used in Sum-Product-Eliminate-Var to create ψ_j, it is removed from the set of factors Φ, and thus cannot be used again. Hence, the cluster graph induced by an execution of variable elimination is necessarily a tree.

We note that although a cluster graph is defined to be an undirected graph, an execution of variable elimination does define a direction for the edges, as induced by the flow of messages between the clusters. The directed graph induced by the messages is a directed tree, with all the messages flowing toward a single cluster where the final result is computed. This cluster is called the *root* of the directed tree. Using standard conventions in computer science, we assume that the root of the tree is "up," so that messages sent toward the root are sent upward. If C_i is on the path from C_j to the root we say that C_i is *upstream* from C_j, and C_j is *downstream* from C_i. We note that, for reasons that will become clear later on, the directions of the edges and the root are not part of the definition of a cluster graph.

upstream clique

downstream clique

The cluster tree defined by variable elimination satisfies an important structural constraint:

Definition 10.2

running intersection property

Let $\mathcal{T}$ be a cluster tree over a set of factors Φ. We denote by $\mathcal{V}_\mathcal{T}$ the vertices of $\mathcal{T}$ and by $\mathcal{E}_\mathcal{T}$ its edges. We say that $\mathcal{T}$ has the running intersection property *if, whenever there is a variable X such that $X \in C_i$ and $X \in C_j$, then X is also in every cluster in the (unique) path in $\mathcal{T}$ between C_i and C_j.* ■

Note that the running intersection property implies that $S_{i,j} = C_i \cap C_j$.

Example 10.2

We can easily check that the running intersection property holds for the cluster tree of figure 10.1. For example, G is present in C_2 and in C_4, so it is also present in the cliques on the path between them: C_3 and C_5. ■

Intuitively, the running intersection property must hold for cluster trees induced by variable elimination because a variable appears in every factor from the moment it is introduced (by multiplying in a factor that mentions it) until it is summed out. We now prove that this property holds in general.

Theorem 10.1

Let $\mathcal{T}$ be a cluster tree induced by a variable elimination algorithm over some set of factors Φ. Then $\mathcal{T}$ satisfies the running intersection property.

Proof Let C and C' be two clusters that contain X. Let C_X be the cluster where X is eliminated. (If X is a query variable, we assume that it is eliminated in the last cluster.) We will prove that X must be present in every cluster on the path between C and C_X, and analogously for C', thereby proving the result.

First, we observe that the computation at C_X must take place later in the algorithm's execution than the computation at C: When X is eliminated in C_X, all of the factors involving X are multiplied into C_X; the result of the summation does not have X in its domain. Hence, after this elimination, Φ no longer has any factors containing X, so no factor generated afterward will contain X in its domain.

By assumption, X is in the domain of the factor in C. We also know that X is not eliminated in C. Therefore, the message computed in C must have X in its domain. By definition, the recipient of X's message, which is C's upstream neighbor in the tree, multiplies in the message from C. Hence, it will also have X in its scope. The same argument applies to show that all cliques upstream from C will have X in their scope, until X is eliminated, which happens only in C_X. Thus, X must appear in all cliques between C and C_X, as required. ■

A very similar proof can be used to show the following result:

Proposition 10.1 *Let $\mathcal{T}$ be a cluster tree induced by a variable elimination algorithm over some set of factors Φ. Let C_i and C_j be two neighboring clusters, such that C_i passes the message τ_i to C_j. Then the scope of the message τ_i is precisely $C_i \cap C_j$.*

The proof is left as an exercise (exercise 10.1).

It turns out that a cluster tree that satisfies the running intersection property is an extremely useful data structure for exact inference in graphical models. We therefore define:

Definition 10.3 clique tree clique *Let Φ be a set of factors over $\mathcal{X}$. A cluster tree over Φ that satisfies the running intersection property is called a* clique tree *(sometimes also called a* junction tree *or a* join tree*). In the case of a clique tree, the clusters are also called* cliques. ■

Note that we have already defined one notion of a clique tree in definition 4.17. This double definition is not an overload of terminology, because the two definitions are actually equivalent: It follows from the results of this chapter that $\mathcal{T}$ is a clique tree for Φ (in the sense of definition 10.3) if and only if it is a clique tree for a chordal graph containing $\mathcal{H}_\Phi$ (in the sense of definition 4.17), and these properties are true if and only if the clique-tree data structure admits variable elimination by passing messages over the tree.

We first show that the running intersection property implies the independence statement, which is at the heart of our first definition of clique trees. Let $\mathcal{T}$ be a cluster tree over Φ, and let $\mathcal{H}_\Phi$ be the undirected graph associated with this set of factors. For any sepset $S_{i,j}$, let $W_{<(i,j)}$ be the set of all variables in the scope of clusters in the C_i side of the tree, and $W_{<(j,i)}$ be the set of all variables in the scope of clusters in the C_j side of the tree.

Theorem 10.2 *$\mathcal{T}$ satisfies the running intersection property if and only if, for every sepset $S_{i,j}$, we have that $W_{<(i,j)}$ and $W_{<(j,i)}$ are separated in $\mathcal{H}_\Phi$ given $S_{i,j}$.*

The proof is left as an exercise (exercise 10.2).

To conclude the proof of the equivalence of the two definitions, it remains only to show that the running intersection property for a tree $\mathcal{T}$ implies that each node in $\mathcal{T}$ corresponds to a clique in a chordal graph $\mathcal{H}'$ containing $\mathcal{H}$, and that each maximal clique in $\mathcal{H}'$ is represented in $\mathcal{T}$. This result follows from our ability to use any clique tree satisfying the running intersection property to perform inference, as shown in this chapter.

10.2 Message Passing: Sum Product

In the previous section, we started out with an execution of the variable elimination algorithm, and showed that it induces a clique tree. In this section, we go in the opposite direction. We assume that we are given a clique tree as a starting point, and we will show how this data structure can be used to perform variable elimination. As we will see, the clique tree is a very

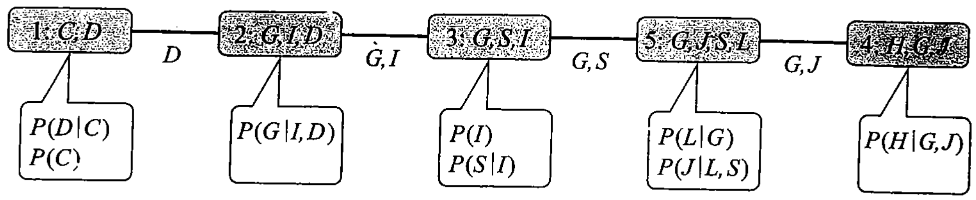


Figure 10.2 Simplified clique tree $\mathcal{T}$ for the Extended Student **network**

useful and versatile data structure. For one, the same clique tree can be used as the basis for many different executions of variable elimination. More importantly, the clique tree provides a data structure for caching computations, allowing multiple executions of variable elimination to be performed much more efficiently than simply performing each one separately.

Consider some set of factors Φ over $\mathcal{X}$, and assume that we are given a clique tree $\mathcal{T}$ over Φ, as defined in definition 4.17. In particular, $\mathcal{T}$ is guaranteed to satisfy the family preservation and running intersection properties. As we now show, we can use the clique tree in several different ways to perform exact inference in graphical models.

10.2.1 Variable Elimination in a Clique Tree

One way of using a clique tree is simply as guidance for the operations of variable elimination. The factors ψ are computed in the cliques, and messages are sent along the edges. Each clique takes the incoming messages (factors), multiplies them, sums out one or more variables, and sends an outgoing message to another clique. As we will see, the clique-tree data structure dictates the operations that are performed on factors in the clique tree and a partial order over these operations. In particular, if clique $\boldsymbol{C}'$ requires a message from $\boldsymbol{C}$, then $\boldsymbol{C}'$ must wait with its computation until $\boldsymbol{C}$ performs its computation and sends the appropriate message to $\boldsymbol{C}'$.

We begin with an example and then describe the general algorithm.

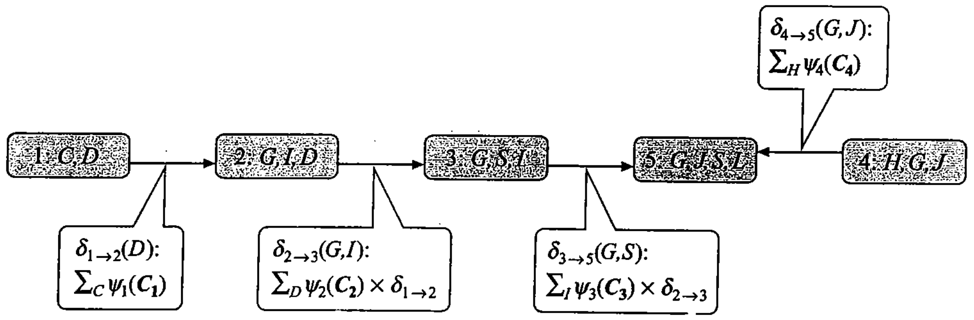
10.2.1.1 An Example

Figure 10.2 shows one possible clique tree $\mathcal{T}$ for the Student network. Note that it is different from the clique tree of figure 10.1, in that nonmaximal cliques ($\boldsymbol{C}_6$ and $\boldsymbol{C}_7$) are absent. Nevertheless, it is straightforward to verify that $\mathcal{T}$ satisfies both the family preservation and the running intersection property. The figure also specifies the assignment α of the initial factors (CPDs) to cliques. Note that, in some cases (for example, the CPD $P(I)$), we have more than one possible clique into which the factor can legally be assigned; as we will see, the algorithm applies for any legal choice.

Our first step is to generate a set of *initial potentials* associated with the different cliques. The initial potential $\psi_i(\boldsymbol{C}_i)$ is computed by multiplying the initial factors assigned to the clique $\boldsymbol{C}_i$. For example, $\psi_5(J, L, G, S) = \phi_L(L, G) \cdot \phi_J(J, L, S)$.

Now, assume that our task is to compute the probability $P(J)$. We want to do the variable elimination process so that J is not eliminated. Thus, we select as our root clique some clique that contains J, for example, $\boldsymbol{C}_5$. We then execute the following steps:

(a)


1: C,D
2: G,I,D
3: G,S,I
5: G,J,S,L
4: H,G,J
$\delta_{1\to2}(D)$: $\sum_C \psi_1(C_1)$
$\delta_{2\to3}(G,I)$: $\sum_D \psi_2(C_2) \times \delta_{1\to2}$
$\delta_{3\to5}(G,S)$: $\sum_I \psi_3(C_3) \times \delta_{2\to3}$
$\delta_{4\to5}(G,J)$: $\sum_H \psi_4(C_4)$

(b)

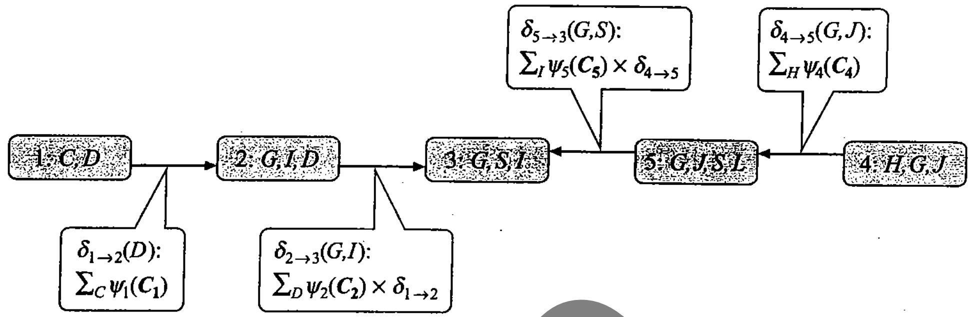

1: C,D
2: G,I,D
3: G,S,I
5: G,J,S,L
4: H,G,J
$\delta_{5\to3}(G,S)$: $\sum_I \psi_5(C_5) \times \delta_{4\to5}$
$\delta_{4\to5}(G,J)$: $\sum_H \psi_4(C_4)$
$\delta_{1\to2}(D)$: $\sum_C \psi_1(C_1)$
$\delta_{2\to3}(G,I)$: $\sum_D \psi_2(C_2) \times \delta_{1\to2}$

Figure 10.3 Two different message propagations with different root cliques in the Student **clique tree:** (a) C_5 is the root; (b) C_3 is the root.

1. **In C_1:** We eliminate C by performing $\sum_C \psi_1(C, D)$. The resulting factor has scope D. We send it as a message $\delta_{1\to2}(D)$ to C_2.
2. **In C_2:** We define $\beta_2(G, I, D) = \delta_{1\to2}(D) \cdot \psi_2(G, I, D)$. We then eliminate D to get a factor over G, I. The resulting factor is $\delta_{2\to3}(G, I)$, which is sent to C_3.
3. **In C_3:** We define $\beta_3(G, S, I) = \delta_{2\to3}(G, I) \cdot \psi_3(G, S, I)$ and eliminate I to get a factor over G, S, which is $\delta_{3\to5}(G, S)$.
4. **In C_4:** We eliminate H by performing $\sum_H \psi_4(H, G, J)$ and send out the resulting factor as $\delta_{4\to5}(G, J)$ to C_5.
5. **In C_5:** We define $\beta_5(G, J, S, L) = \delta_{3\to5}(G, S) \cdot \delta_{4\to5}(G, J) \cdot \psi_5(G, J, S, L)$.

The factor β_5 is a factor over G, J, S, L that encodes the joint distribution $P(G, J, L, S)$: all the CPDs have been multiplied in, and all the other variables have been eliminated. If we now want to obtain $P(J)$, we simply sum out G, L, and S.

We note that the operations in the elimination process could also have been done in another order. The only constraint is that a clique get all of its incoming messages from its downstream neighbors before it sends its outgoing message toward its upstream neighbor. We say that a clique is *ready* when it has received all of its incoming messages. Thus, for example, C_4 is ready

ready clique

at the very start of the algorithm, and the computation associated with it can be performed at any point in the execution. However, C_2 is ready only after it receives its message from C_1. Thus, C_1, C_4, C_2, C_3, C_5 is a legal execution ordering for a tree rooted at C_5, whereas C_2, C_1, C_4, C_3, C_5 is not. Overall, the set of messages transmitted throughout the execution of the algorithm is shown in figure 10.3a.

As we mentioned, the choice of root clique is not fully determined. To derive $P(J)$, we could have chosen C_4 as the root. Let us see how the algorithm would have changed in that case:

1. **In C_1:** The computation and message are unchanged.
2. **In C_2:** The computation and message are unchanged.
3. **In C_3:** The computation and message are unchanged.
4. **In C_5:** We define $\beta_5(G, J, S, L) = \delta_{3\to5}(G, S) \cdot \psi_5(G, J, S, L)$ and eliminate S and L. We send out the resulting factor as $\delta_{5\to4}(G, J)$ to C_4.
5. **In C_4:** We define $\beta_4(H, G, J) = \delta_{5\to4}(G, S) \cdot \psi_4(H, G, J)$.

We can now extract $P(J)$ by eliminating H and G from $\beta_4(H, G, J)$.

In a similar way, we can apply exactly the same process to computing the distribution over any other variable. For example, if we want to compute the probability $P(G)$, we could choose any of the cliques where it appears. If we use C_3, for example, the computation in C_1 and C_2 is identical. The computation in C_4 is the same as in the first of our two executions: a message is computed and sent to C_5. In C_5, we compute $\beta_5(G, J, S, L) = \delta_{4\to5}(G, J) \cdot \psi_5(G, J, S, L)$, and we eliminate G and L to produce a message $\delta_{5\to3}(G, S)$, which can then be sent to C_3 and used in the operation:

$$\beta_3(G, S, I) = \delta_{2\to3}(G, I) \cdot \delta_{5\to3}(G, S) \cdot \psi_3(G, S, I).$$

Overall, the set of messages transmitted throughout this execution of the algorithm is shown in figure 10.3b.

10.2.1.2 Clique-Tree Message Passing

message passing

We can now specify a general variable elimination algorithm that can be implemented via *message passing* in a clique tree. Let $\mathcal{T}$ be a clique tree with the cliques $C_1, \ldots, C_k$. We begin by multiplying the factors assigned to each clique, resulting in our initial potentials. We then use the clique-tree data structure to pass messages between neighboring cliques, sending all messages toward the root clique. We describe the algorithm in abstract terms; box 10.A provides some important tips for efficient implementation.

initial potential

Recall that each factor $\phi \in \Phi$ is assigned to some clique $\alpha(\phi)$. We define the *initial potential* of C_j to be:

$$\psi_j(C_j) = \prod_{\phi\ :\ \alpha(\phi)=j} \phi.$$

Because each factor is assigned to exactly one clique, we have that

$$\prod_{\phi} \phi = \prod_{j} \psi_j.$$

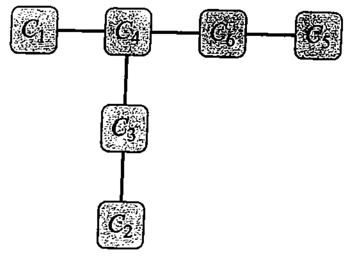


Figure 10.4 An abstract clique tree that is not chain-structured

Let C_r be the selected root clique. We now perform sum-product variable elimination over the cliques, starting from the leaves of the clique tree and moving inward. More precisely, for each clique C_i, we define Nb_i to be the set of indexes of cliques that are neighbors of C_i. Let $p_r(i)$ be the upstream neighbor of i (the one on the path to the root clique r). Each clique C_i, except for the root, performs a message passing computation and sends a message to its upstream neighbor $C_{p_r(i)}$.

sum-product message passing

The message from C_i to another clique C_j is computed using the following *sum-product message passing* computation:

$$\delta_{i\to j} = \sum_{C_i - S_{i,j}} \psi_i \cdot \prod_{k \in (\mathrm{Nb}_i - \{j\})} \delta_{k\to i}. \tag{10.2}$$

In words, the clique C_i multiplies all incoming messages from its other neighbors with its initial clique potential, resulting in a factor ψ whose scope is the clique. It then sums out all variables except those in the sepset between C_i and C_j, and sends the resulting factor as a message to C_j.

beliefs

This message passing process proceeds up the tree, culminating at the root clique. When the root clique has received all messages, it multiplies them with its own initial potential. The result is a factor called the *beliefs*, denoted $\beta_r(C_r)$. It represents, as we show,

$$\tilde{P}_\Phi(C_r) = \sum_{\mathcal{X} - C_r} \prod_{\phi} \phi.$$

The complete algorithm is shown in algorithm 10.1.

Example 10.3

Consider the abstract clique tree of figure 10.4, and assume that we have selected C_6 as our root clique. The numbering of the cliques denotes one possible ordering of the operations, with C_1 being the first to compute its message. However, multiple other orderings are legitimate, for example, $2, 5, 1, 3, 4, 6$; in general, any ordering that respects the ordering constraints $\{(2 \prec 3), (3 \prec 4), (1 \prec 4), (4 \prec 6), (5 \prec 6)\}$ is a legal ordering for the message passing process. ■

We can use this algorithm to compute the marginal probability of any set of query nodes Y which is fully contained in some clique. We select one such clique C_r to be the root, and perform the clique-tree message passing toward that root. We then extract $\tilde{P}_\Phi(Y)$ from the final potential at C_r by summing out the other variables $C_r - Y$.

Algorithm 10.1 Upward pass of variable elimination in clique tree

```
Procedure CTree-SP-Upward (
   $\Phi$,   // Set of factors
   $\mathcal{T}$,   // Clique tree over $\Phi$
   $\alpha$,   // Initial assignment of factors to cliques
   $C_r$   // Some selected root clique
  )
1   Initialize-Cliques
2   while $C_r$ is not ready
3     Let $C_i$ be a ready clique
4     $\delta_{i \to p_r(i)}(S_{i,p_r(i)}) \leftarrow$ SP-Message$(i, p_r(i))$
5   $\beta_r \leftarrow \psi_r \cdot \prod_{k \in \mathrm{Nb}_{C_r}} \delta_{k \to r}$
6   return $\beta_r$

Procedure Initialize-Cliques (

  )
1   for each clique $C_i$
2     $\psi_i(C_i) \leftarrow \prod_{\phi_j \,:\, \alpha(\phi_j)=i} \phi$
3

Procedure SP-Message (
   i,   // sending clique
   j   // receiving clique
  )
1   $\psi(C_i) \leftarrow \psi_i \cdot \prod_{k \in (\mathrm{Nb}_i - \{j\})} \delta_{k \to i}$
2   $\tau(S_{i,j}) \leftarrow \sum_{C_i - S_{i,j}} \psi(C_i)$
3   return $\tau(S_{i,j})$
```

10.2.1.3 Correctness

We now prove that this algorithm, when applied to a clique tree that satisfies the family preservation and running intersection property, computes the desired expressions over the messages and the cliques.

In our algorithm, a variable X is eliminated only when a message is sent from C_i to a neighboring C_j such that $X \in C_i$ and $X \notin C_j$. We first prove the following result:

Proposition 10.2 *Assume that X is eliminated when a message is sent from C_i to C_j. Then X does not appear anywhere in the tree on the C_j side of the edge $(i\text{–}j)$.*

Proof The proof is a simple consequence of the running intersection property. Assume by contradiction that X appears in some other clique C_k that is on the C_j side of the tree. Then C_j is on the path from C_i to C_k. But we know that X appears in both C_i and C_k but not in C_j, violating the running intersection property. ■

Based on this result, we can provide a semantic interpretation for the messages used in the clique tree. Let $(i–j)$ be some edge in the clique tree. We use $\mathcal{F}_{\prec(i\rightarrow j)}$ to denote the set of factors in the cliques on the $\boldsymbol{C}_i$-side of the edge and $\mathcal{V}_{\prec(i\rightarrow j)}$ to denote the set of variables that appear on the $\boldsymbol{C}_i$-side but are not in the sepset. For example, in the clique tree of figure 10.2, we have that $\mathcal{F}_{\prec(3\rightarrow 5)} = \{P(C), P(D \mid C), P(G \mid I, D), P(I), P(S \mid I)\}$ and $\mathcal{V}_{\prec(3\rightarrow 5)} = \{C, D, I\}$. Intuitively, the message passed between the cliques $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ is the product of all the factors in $\mathcal{F}_{\prec(i\rightarrow j)}$, marginalized over the variables in the sepset (that is, summing out all the others).

Theorem 10.3

Let $\delta_{i\rightarrow j}$ be a message from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$. Then:

$$\delta_{i\rightarrow j}(\boldsymbol{S}_{i,j}) = \sum_{\mathcal{V}_{\prec(i\rightarrow j)}} \prod_{\phi\in\mathcal{F}_{\prec(i\rightarrow j)}} \phi.$$

Proof The proof proceeds by induction on the length of the path from the leaves. For the base case, the clique $\boldsymbol{C}_i$ is a leaf in the tree. In this case, the result follows from a simple examination of the operations executed at the clique.

Now, consider a clique $\boldsymbol{C}_i$ that is not a leaf, and consider the expression

$$\sum_{\mathcal{V}_{\prec(i\rightarrow j)}} \prod_{\phi\in\mathcal{F}_{\prec(i\rightarrow j)}} \phi. \tag{10.3}$$

Let $i_1, \ldots, i_m$ be the neighboring cliques of $\boldsymbol{C}_i$ other than $\boldsymbol{C}_j$. It follows immediately from proposition 10.2 that $\mathcal{V}_{\prec(i\rightarrow j)}$ is the disjoint union of $\mathcal{V}_{\prec(i_k\rightarrow i)}$ for $k = 1, \ldots, m$ and the variables $\boldsymbol{Y}_i$ eliminated at $\boldsymbol{C}_i$ itself. Similarly, $\mathcal{F}_{\prec(i\rightarrow j)}$ is the disjoint union of the $\mathcal{F}_{\prec(i_k\rightarrow i)}$ and the factors $\mathcal{F}_i$ from which ψ_i was computed. Thus equation (10.3) is equal to

$$\sum_{\boldsymbol{Y}_i} \sum_{\mathcal{V}_{\prec(i_1\rightarrow i)}} \cdots \sum_{\mathcal{V}_{\prec(i_m\rightarrow i)}} \left(\prod_{\phi\in\mathcal{F}_{\prec(i_1\rightarrow i)}} \phi\right) \cdot \cdots \cdot \left(\prod_{\phi\in\mathcal{F}_{\prec(i_m\rightarrow i)}} \phi\right) \cdot \left(\prod_{\phi\in\mathcal{F}_i} \phi\right). \tag{10.4}$$

As we just showed, for each k, none of the variables in $\mathcal{V}_{\prec(i_k\rightarrow i)}$ appear in any of the other factors. Thus, we can use equation (9.6) and push in the summation over $\mathcal{V}_{\prec(i_k\rightarrow i)}$ in equation (10.4), and obtain:

$$\sum_{\boldsymbol{Y}_i} \left(\prod_{\phi\in\mathcal{F}_i} \phi\right) \cdot \sum_{\mathcal{V}_{\prec(i_1\rightarrow i)}} \left(\prod_{\phi\in\mathcal{F}_{\prec(i_1\rightarrow i)}} \phi\right) \cdot \cdots \cdot \sum_{\mathcal{V}_{\prec(i_m\rightarrow i)}} \left(\prod_{\phi\in\mathcal{F}_{\prec(i_m\rightarrow i)}} \phi\right). \tag{10.5}$$

Using the inductive hypothesis and the definition of ψ_i, this expression is equal to

$$\sum_{\boldsymbol{Y}_i} \psi_i \cdot \delta_{i_1\rightarrow i} \cdot \cdots \cdot \delta_{i_m\rightarrow i}, \tag{10.6}$$

which is precisely the operation used to compute the message $\delta_{i\rightarrow j}$. ■

This theorem is closely related to theorem 10.2, which tells us that a sepset divides the graph into conditionally independent pieces. It is this conditional independence property that allows the message over the sepset to summarize completely the information in one side of the clique tree that is necessary for the computation in the other.

Based on this analysis, we can show that:

Corollary 10.1 *Let C_r be the root clique in a clique tree, and assume that β_r is computed as in the algorithm of algorithm 10.1. Then*

$$\beta_r(C_r) = \sum_{\mathcal{X}-C_r} \tilde{P}_\Phi(\mathcal{X}).$$

As we discussed earlier, this algorithm applies both to Bayesian network and Markov network inference. For a Bayesian network $\mathcal{B}$, if Φ consists of the CPDs in $\mathcal{B}$, reduced with some evidence e, then $\beta_r(C_r) = P_\mathcal{B}(C_r, e)$. For a Markov network $\mathcal{H}$, if Φ consists of the compatibility functions defining the network, then $\beta_r(C_r) = \tilde{P}_\Phi(C_r)$. In both cases, we can obtain the probability over the variables in C_r as usual, by normalizing the resulting factor to sum to 1. In the Markov network, we can also obtain the value of the partition function simply by summing up all of the entries in the potential of the root clique $\beta_r(C_r)$.

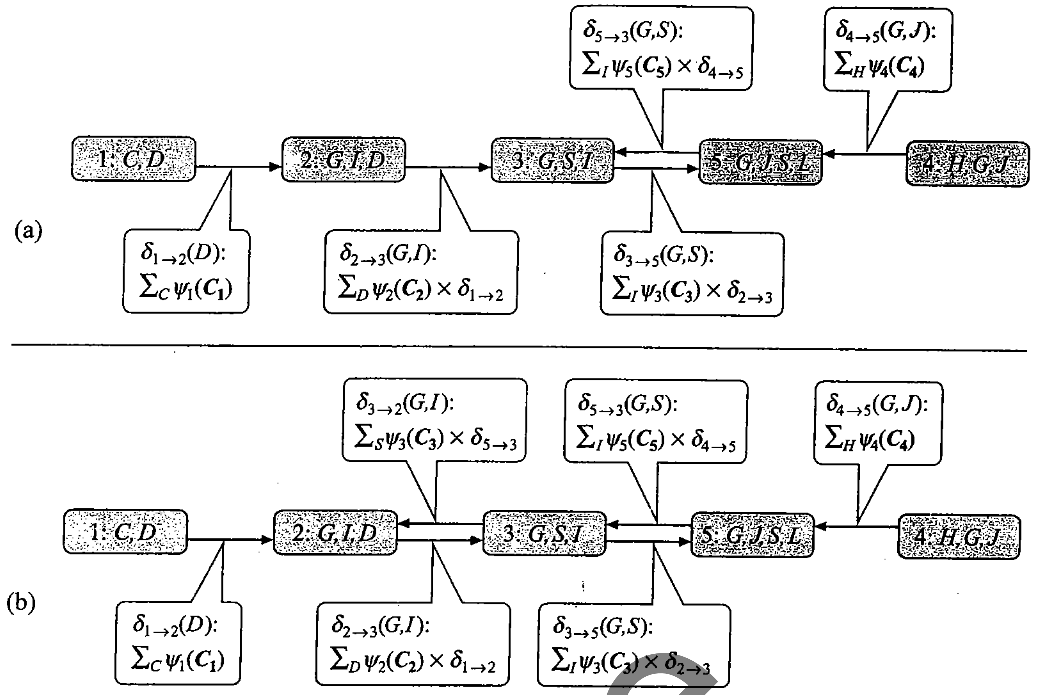
10.2.2 Clique Tree Calibration

We have shown that we can use the same clique tree to compute the probability of any variable in $\mathcal{X}$. In many applications, we often wish to estimate the probability of a large number of variables. For example, in a medical-diagnosis setting, we generally want the probability of several possible diseases. Furthermore, as we will see, when learning Bayesian networks from partially observed data, we always want the probability distributions over each of the unobserved variables in the domain (and their parents).

Therefore, let us consider the task of computing the posterior distribution over every random variable in the network. The most naive approach is to do inference separately for each variable. Letting c be the cost of a single execution of clique tree inference, the total cost of this algorithm is nc. An approach that is slightly less naive is to run the algorithm once for every clique, making it the root. The total cost of this variant is Kc, where K is the number of cliques. However, it turns out that we can do substantially better than either of these approaches.

Let us revisit our clique tree of figure 10.2 and consider the three different executions of the clique tree algorithm that we described: one where C_5 is the root, one where C_4 is the root, and one where C_3 is the root. As we pointed out, the messages sent from C_1 to C_2 and from C_2 to C_3 are the same in all three executions. The message sent from C_4 to C_5 is the same in both of the executions where it appears. In the second of the two executions, there simply is no message from C_4 to C_5 — the message goes the other way, from C_5 to C_4.

More generally, consider two neighboring cliques C_i and C_j in some clique tree. It follows from theorem 10.3 that the value of the message sent from C_i to C_j does not depend on specific choice of root clique: As long as the root clique is on the C_j-side, exactly the same message is sent from C_i to C_j. The same argument applies if the root is on the C_i-side. Thus, in all executions of the clique tree algorithm, whenever a message is sent between two cliques, it is necessarily the same. Thus, for any given clique tree, each edge has two messages associated with it: one for each direction of the edge. If we have a total of c cliques, there are $c-1$ edges in the tree; therefore, we have $2(c-1)$ messages to compute.



$\delta_{5\to3}(G,S)$: $\sum_I \psi_5(C_5) \times \delta_{4\to5}$
$\delta_{4\to5}(G,J)$: $\sum_H \psi_4(C_4)$
1: C,D
2: G,I,D
3: G,S,I
5: G,J,S,L
4: H,G,J
(a)
$\delta_{1\to2}(D)$: $\sum_C \psi_1(C_1)$
$\delta_{2\to3}(G,I)$: $\sum_D \psi_2(C_2) \times \delta_{1\to2}$
$\delta_{3\to5}(G,S)$: $\sum_I \psi_3(C_3) \times \delta_{2\to3}$
$\delta_{3\to2}(G,I)$: $\sum_S \psi_3(C_3) \times \delta_{5\to3}$
$\delta_{5\to3}(G,S)$: $\sum_I \psi_5(C_5) \times \delta_{4\to5}$
$\delta_{4\to5}(G,J)$: $\sum_H \psi_4(C_4)$
1: C,D
2: G,I,D
3: G,S,I
5: G,J,S,L
4: H,G,J
(b)
$\delta_{1\to2}(D)$: $\sum_C \psi_1(C_1)$
$\delta_{2\to3}(G,I)$: $\sum_D \psi_2(C_2) \times \delta_{1\to2}$
$\delta_{3\to5}(G,S)$: $\sum_I \psi_3(C_3) \times \delta_{2\to3}$

Figure 10.5 Two steps in a downward pass in the Student network

We can compute both messages for each edge by the following simple asynchronous algorithm. Recall that a clique can transmit a message upstream toward the root when it has all of the messages from its downstream neighbors. We can generalize this concept as follows:

Definition 10.4 ready clique

Let $\mathcal{T}$ be a clique tree. We say that C_i is ready *to transmit to a neighbor C_j when C_i has messages from all of its neighbors except from C_j.* ■

When C_i is ready to transmit to C_j, it can compute the message $\delta_{i\to j}(S_{i,j})$ by multiplying its initial potential with all of its incoming messages except the one from C_j, and then eliminate the variables in $C_i - S_{i,j}$. In effect, this algorithm uses yet another layer of *dynamic programming* to avoid recomputing the same message multiple times.

dynamic programming

sum-product belief propagation

Algorithm 10.2 shows the full procedure, often called *sum-product belief propagation.* As written, the algorithm is defined *asynchronously*, with each clique sending a message as soon as it is ready. One might wonder why this process is guaranteed to terminate, that is, why there is always a clique that is ready to transmit to some other clique. In fact, the message passing process performed by the algorithm is equivalent to a much more systematic process that consists of an *upward pass* and a *downward pass*. In the upward pass, we first pick a root and send all messages toward the root. When this process is complete, the root has all messages. Therefore, it can now send the appropriate message to all of its children. This

upward pass

downward pass

Algorithm 10.2 Calibration using sum-product message passing in a clique tree

```
Procedure CTree-SP-Calibrate (
    $\Phi$,  // Set of factors
    $\mathcal{T}$  // Clique tree over $\Phi$
  )
1   Initialize-Cliques
2   while exist $i, j$ such that $i$ is ready to transmit to $j$
3     $\delta_{i \to j}(S_{i,j}) \leftarrow$ SP-Message$(i, j)$
4   for each clique $i$
5     $\beta_i \leftarrow \psi_i \cdot \prod_{k \in \mathrm{Nb}_i} \delta_{k \to i}$
6   return $\{\beta_i\}$
```

algorithm continues until the leaves of the tree are reached, at which point no more messages need to be sent. This second phase is called the downward pass. The asynchronous algorithm is equivalent to this systematic algorithm, except that the root is simply the first clique that happens to obtain messages from all of its neighbors. In an actual implementation, we might want to *schedule* this process more explicitly. (At the very least, the algorithm would check in line 2 that a message is not computed more than once.)

message scheduling

Example 10.4

Figure 10.3a shows the upward pass of the clique tree algorithm when C_5 is the root. Figure 10.5a shows a possible first step in a downward pass, where C_5 sends a message to its child C_3, based on the message from C_4 and its initial potential. As soon as a child of the root receives a message, it has all of the information it needs to send a message to its own children. Figure 10.5b shows C_3 sending the downward message to C_2. ■

beliefs

At the end of this process, we compute the *beliefs* for all cliques in the tree by multiplying the initial potential with each of the incoming messages. The key is to note that the messages used in the computation of β_i are precisely the same messages that would have been used in a standard upward pass of the algorithm with C_i as the root. Thus, we conclude:

Corollary 10.2

Assume that, for each clique i, β_i is computed as in the algorithm of algorithm 10.2. Then

$$\beta_i(C_i) = \sum_{\mathcal{X} - C_i} \tilde{P}_\Phi(\mathcal{X}).$$

Note that it is important that C_i compute the message to a neighboring clique C_j based on its *initial potential* ψ_i and not its modified potential β_i. The latter already integrates information from j. If the message were computed based on this latter potential, we would be double-counting the factors assigned to C_j (multiplying them twice into the joint).

When this process concludes, each clique contains the marginal (unnormalized) probability over the variables in its scope. As we discussed, we can compute the marginal probability over a particular variable X by selecting a clique whose scope contains X, and eliminating the redundant variables in the clique. A key point is that the result of this process does not depend on the clique we selected. That is, if X appears in two cliques, they must agree on its marginal.

Definition 10.5

calibrated

Two adjacent cliques C_i and C_j are said to be calibrated *if*

$$\sum_{C_i - S_{i,j}} \beta_i(C_i) = \sum_{C_j - S_{i,j}} \beta_j(C_j).$$

beliefs

A clique tree $\mathcal{T}$ is calibrated *if all pairs of adjacent cliques are calibrated. For a calibrated clique tree, we use the term* clique beliefs *for $\beta_i(C_i)$ and* sepset beliefs *for*

$$\mu_{i,j}(S_{i,j}) = \sum_{C_i - S_{i,j}} \beta_i(C_i) = \sum_{C_j - S_{i,j}} \beta_j(C_j). \tag{10.7}$$

■

The main advantage of the clique tree algorithm is that it computes the posterior probability of all variables in a graphical model using only twice the computation of the upward pass in the same tree. Letting c once again be the execution cost of message passing in a clique tree to one root, the cost of this algorithm is $2c$. By comparison, recall that the cost of doing a separate computation for each variable is nc and a separate computation for each root clique is Kc, where K is the number of cliques. In most cases, the savings are considerable, making the clique tree algorithm the algorithm of choice in situations where we want to compute the posterior of multiple query variables.

Box 10.A — Skill: Efficient Implementation of Factor Manipulation Algorithms. *While simple conceptually, the implementation of algorithms involving manipulation of factors can be surprisingly subtle. In particular,* **different design decisions can lead to orders-of-magnitude differences in performance, as well as differences in the accuracy of the results.** *We now discuss some of the key design decisions in these algorithms. We note that the methods we describe here are equally applicable to the algorithms in many of the other chapters in the book, including the variable elimination algorithm of chapter 9, the exact and approximate sum-product message passing algorithms of chapters 10 and 11, and many of the MAP algorithms of chapter 13.*

The first key decision is the representation of our basic data structure: a factor, or a multidimensional table, with an entry for each possible assignment to the variables. One standard technique for storing multidimensional tables is to flatten them into a single array in computer memory. For each variable, we also store its cardinality, and its stride, *or step size in the factor. For example, given a factor $\phi(A, B, C)$ over variables A, B, and C, with cardinalities 2, 2, and 3, respectively, we can represent the factor in memory by the array*

stride

$$\texttt{phi}[0\ldots 11] = \{\phi(a^1,b^1,c^1), \phi(a^2,b^1,c^1), \phi(a^1,b^2,c^1), \ldots, \phi(a^2,b^2,c^3)\}.$$

Here the stride for variable A is 1, for B is 2 and for C is 4. If we add a fourth variable, D, its stride would be 12, since we would need to step over twelve entries before reaching the next assignment to D. Notice how, using each variable's stride, we can easily go from a variable assignment to a corresponding index into the factor array

$$\texttt{index} = \sum_i \texttt{assignment}[i] \cdot \texttt{phi.stride}[i]$$

Algorithm 10.A.1 — Efficient implementation of a factor product operation.

```
Procedure Factor-Product (
   phi1 over scope $X_1$,
  phi2 over scope $X_2$
    // Factors represented as a flat array with strides for the variables
)
  $j \leftarrow 0, k \leftarrow 0$
  for $l = 0, \dots, |X_1 \cup X_2|$
    assignment[$l$] $\leftarrow 0$
  for $i = 0, \dots, |Val(X_1 \cup X_2)| - 1$
    psi[$i$] $\leftarrow$ phi1[$j$] $\cdot$ phi2[$k$]
    for $l = 0, \dots, |X_1 \cup X_2|$
      assignment[$l$] $\leftarrow$ assignment[$l$] $+ 1$
      if assignment[$l$] $=$ card[$l$] then
        $j \leftarrow j - ($card[$l$]$ - 1) \cdot$ phi1.stride[$l$]
        $k \leftarrow k - ($card[$l$]$ - 1) \cdot$ phi2.stride[$l$]
      else
        $j \leftarrow j +$ phi1.stride[$l$]
        $k \leftarrow k +$ phi2.stride[$l$]
        break
  return (psi)
```

and vice versa

$$\texttt{assignment}[i] = \lfloor \texttt{index}/\texttt{phi.stride}[i] \rfloor \bmod \texttt{card}[i]$$

With this factor representation, we can now design a library of operations: product, marginalization, maximization, reduction, *and so forth. Since many inference algorithms involve multiple iterations over a series of factor operations, it is important that these be high-performance. One of the key design decisions is indexing the appropriate entries in each factor for the operations that we wish to perform. (In fact, when one uses a naive implementation of index computations, one often discovers that 90 percent of the running time is spent on that task.)*

factor product

algorithm 10.A.1 provides an example for the product *between two arbitrary factors. Here we define* `phi.stride`$[X] = 0$ *if* $X \notin Scope[\phi]$. *The inner loop (over l) advances to the next assignment to the variables in* ψ *and calculates indexes into each other factor array on the fly. It can be understood by considering the equation for computing* `index` *shown earlier. Similar on-the-fly index calculations can be applied for other factor operations. We leave these as an exercise (exercise 10.3).*

For iterative algorithms or multiple queries, where the same operation (on different data) is performed a large number of times, it may be beneficial to cache these index mappings for later use. Note, however, that the index mappings require the same amount of storage as the factors themselves, that is, are exponential in the number of variables. Thus, this design choice offers a direct trade-off between memory usage and speed, especially in view of the fact that the index *computations require approximately the same amount of work as the factor operation itself. Since performance of main memory is orders of magnitudes faster than secondary storage (disk), when memory limitations are an issue, it is better not to cache index mappings for blarge problems. One exception is template models, where savings can be made by reusing the same indexes for different instantiations of the factor templates.*

An additional trick in reducing the computational burden is to preallocate and reuse memory for factor storage. Allocating memory is a relatively expensive procedure, and one does not want to waste time on this task inside a tight loop. To illustrate this point, we consider the example of variable elimination for computing $\psi(A, D)$ *as*

$$\psi(A, D) = \sum_{B,C} \phi_1(A, B)\phi_2(B, C)\phi_3(C, D) = \sum_{B} \phi_1(A, B) \sum_{C} \phi_2(B, C)\phi_3(C, D).$$

Here we need to compute three intermediate factors: $\tau_1(B, C, D) = \phi_2(B, C)\phi_3(C, D)$; $\tau_2(B, D) = \sum_C \tau_1(B, C, D)$; *and* $\tau_3(A, B, D) = \phi_1(A, B)\tau_2(B, D)$. *Notice that, once* $\tau_2(B, D)$ *has been calculated, we no longer need the values in* $\tau_1(B, C, D)$. *By initially allocating memory large enough to hold the larger of* $\tau_1(B, C, D)$ *and* $\tau_3(A, B, D)$, *we can use the same memory for both. Because every operation in a variable elimination or message passing algorithm requires the computation of one or more intermediate factors, some of which are much larger than the desired end product, the savings in both time (preallocation) and memory (reusage) can be significant.*

We now turn our attention to numerical considerations. Operations such as factor product involve multiplying many small numbers together, which can lead to underflow problems due to finite precision arithmetic. The problem can be alleviated somewhat by renormalizing the factor after each operation (so that the maximum entry in the factor is one); this operation leaves the results to most queries unchanged (see exercise 9.3). However, if each entry in the factor is computed as the product of many terms, underflow can still occur. An alternative solution is to perform the computation in log-space, *replacing multiplications with additions; this transformation allows for greater machine precision to be utilized. Note that* marginalization, *which requires that we sum entries, cannot be performed in log-space; it requires exponentiating each entry in the factor, performing the marginalization, and taking the log of the result. Since moving from log-space to probability-space incurs a significant decrease in dynamic range, factors should be normalized before applying this transform. One standard trick is to shift every entry by the maximum entry*

log-space

factor marginalization

$$\texttt{phi}[i] \leftarrow \exp\{\texttt{logPhi}[i] - c\},$$

where $c = \max_i \texttt{logPhi}[i]$; *this transformation ensures that the resulting factor has a maximum entry of one and prevents overflow.*

We note that there are some caveats to operating in log-space. First, one may incur a performance hit: Floating point multiplication is no slower than floating point addition, but the transformation to and from log-space, as required for marginalization, can take a significant proportion of the total processing time. This caveat does not apply to algorithms such as max-product, where maximization can be performed in log-space; indeed, these algorithms are almost always implemented as max-sum. Moreover, log-space operations require care in handling nonpositive factors (that is, factors with some zero entries).

Finally, at a higher level, as with any software implementation, there is always a trade-off between speed, memory consumption, and reusability of the code. For example, software specialized for the case of pairwise potentials over a grid will almost certainly outperform code written for general *graphs with arbitrary potentials. However, the small performance hit in using well designed general purpose code often outweighs the development effort required to reimplement algorithms for each specialized application. However, as always, it is also important not to try to optimize code too early. It is more beneficial to write and profile the code, on real examples, to determine what operations are causing bottlenecks. This allows the development effort to be targeted to areas that can yield the most gain.*

10.2.3 A Calibrated Clique Tree as a Distribution

A calibrated clique tree is more than simply a data structure that stores the results of probabilistic inference for all of the cliques in the tree. As we now show, it can also be viewed as an alternative representation of the measure $\tilde{P}_\Phi$.

At calibration, we have that:

$$\beta_i = \psi_i \cdot \prod_{k \in \mathrm{Nb}_i} \delta_{k \to i}. \tag{10.8}$$

We also have that:

$$\begin{aligned}
\mu_{i,j}(\boldsymbol{S}_{i,j}) &= \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{C}_i) \\
&= \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in \mathrm{Nb}_i} \delta_{k \to i} \\
&= \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \delta_{j \to i} \prod_{k \in (\mathrm{Nb}_i - \{j\})} \delta_{k \to i} \\
&= \delta_{j \to i} \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in (\mathrm{Nb}_i - \{j\})} \delta_{k \to i} \\
&= \delta_{j \to j} \delta_{i \to j},
\end{aligned} \tag{10.9}$$

where the fourth equality holds because no variable in the scope of $\delta_{j \to i}$ is involved in the summation.

We can now show the following important result:

Proposition 10.3 *At convergence of the clique tree calibration algorithm, we have that:*

$$\tilde{P}_\Phi(\mathcal{X}) = \frac{\prod_{i \in \mathcal{V}_\mathcal{T}} \beta_i(\boldsymbol{C}_i)}{\prod_{(i-j) \in \mathcal{E}_\mathcal{T}} \mu_{i,j}(\boldsymbol{S}_{i,j})}. \tag{10.10}$$

Proof Using equation (10.8), the numerator in the right-hand side of equation (10.10) can be rewritten as:

$$\prod_{i \in \mathcal{V}_\mathcal{T}} \psi_i(\boldsymbol{C}_i) \prod_{k \in \mathrm{Nb}_i} \delta_{k \to i}.$$

Assignment			$\max_C$
a^0	b^0	d^0	$600,000$
a^0	b^0	d^1	$300,030$
a^0	b^1	d^0	$5,000,500$
a^0	b^1	d^1	$1,000$
a^1	b^0	d^0	200
a^1	b^0	d^1	$1,000,100$
a^1	b^1	d^0	$100,010$
a^1	b^1	d^1	$200,000$

$\beta_1(A, B, D)$

Assignment		$\max_{A,C}$
b^0	d^0	$600,200$
b^0	d^1	$1,300,130$
b^1	d^0	$5,100,510$
b^1	d^1	$201,000$

$\mu_{1,2}(B, D)$

Assignment			$\max_A$
b^0	c^0	d^0	$300,100$
b^0	c^0	d^1	$1,300,000$
b^0	c^1	d^0	$300,100$
b^0	c^1	d^1	130
b^1	c^0	d^0	510
b^1	c^0	d^1	$100,500$
b^1	c^1	d^0	$5,100,000$
b^1	c^1	d^1	$100,500$

$\beta_2(B, C, D)$

Figure 10.6 The clique and sepset beliefs for the Misconception **example.**

Using equation (10.9), the denominator can be rewritten as:

$$\prod_{(i-j)\in\mathcal{E}_\mathcal{T}} \delta_{i\to j}\delta_{j\to i}.$$

Each message $\delta_{i\to j}$ appears exactly once in the numerator and exactly once in the denominator, so that all messages cancel. The remaining expression is simply:

$$\prod_{i\in\mathcal{V}_\mathcal{T}} \psi_i(\boldsymbol{C}_i) = \tilde{P}_\Phi. \quad ■$$

reparameterization

clique tree invariant

Thus, via equation (10.10), the clique and sepsets beliefs provide a *reparameterization* of the unnormalized measure. This property is called the *clique tree invariant*, for reasons which will become clear later on in this chapter.

Another intuition for this result can be obtained from the following example:

Example 10.5

Consider a clique tree obtained from Markov network $A—B—C—D$, with an appropriate set of factors Φ. Our clique tree in this case would have three cliques $\boldsymbol{C}_1 = \{A, B\}$, $\boldsymbol{C}_2 = \{B, C\}$, and $\boldsymbol{C}_3 = \{C, D\}$. When the clique tree is calibrated, we have that $\beta_1(A, B) = \tilde{P}_\Phi(A, B)$ and $\beta_2(B, C) = \tilde{P}_\Phi(B, C)$. From the conditional independence properties of this distribution, we have that

$$\tilde{P}_\Phi(A, B, C) = \tilde{P}_\Phi(A, B)\tilde{P}_\Phi(C \mid B),$$

and

$$\tilde{P}_\Phi(C \mid B) = \frac{\beta_2(B, C)}{\tilde{P}_\Phi(B)}.$$

As $\beta_2(B, C) = \tilde{P}_\Phi(B, C)$, we can obtain $\tilde{P}_\Phi(B)$ by marginalizing $\beta_2(B, C)$. Thus, we can write:

$$\begin{aligned}\tilde{P}_\Phi(A, B, C) &= \beta_1(A, B)\frac{\beta_2(B, C)}{\sum_C \beta_2(B, C)} \\ &= \frac{\beta_1(A, B)\beta_2(B, C)}{\sum_C \beta_2(B, C)}.\end{aligned}$$

In fact, when the two cliques are calibrated, they must agree on the marginal of B. Thus, the expression in the denominator can equivalently be replaced by $\sum_A \beta_1(A,B)$. ■

Based on this analysis, we now formally define the distribution represented by a clique tree:

Definition 10.6

clique tree measure

We define the measure *induced by a calibrated tree $\mathcal{T}$ to be:*

$$Q_{\mathcal{T}} = \frac{\prod_{i \in \mathcal{V}_{\mathcal{T}}} \beta_i(C_i)}{\prod_{(i-j) \in \mathcal{E}_{\mathcal{T}}} \mu_{i,j}(S_{i,j})}, \tag{10.11}$$

where

$$\mu_{i,j} = \sum_{C_i - S_{i,j}} \beta_i(C_i) = \sum_{C_j - S_{i,j}} \beta_j(C_j).$$ ■

Example 10.6

Consider, for example, the Markov network of example 3.8, whose joint distribution is shown in figure 4.2. One clique tree for this network consists of the two cliques $\{A,B,D\}$ and $\{B,C,D\}$, with the sepset $\{B,D\}$. The final potentials and sepset for this example are shown in figure 10.6. It is straightforward to confirm that the clique tree is indeed calibrated. One can also verify that this clique tree provides a reparameterization of the original distribution. For example, consider the entry $\tilde{P}_\Phi(a^1,b^0,c^1,d^0) = 100$. According to equation (10.10), the clique tree measure is:

$$\frac{\beta_1(a^1,b^0,d^0)\beta_2(b^0,c^1,d^0)}{\mu_{1,2}(b^0,d^0)} = \frac{200 \cdot 300,100}{600,200} = 100,$$

as required. ■

Our analysis so far shows that for a set of calibrated potentials derived from clique tree inference, we have two properties: the clique tree measure is $\tilde{P}_\Phi$ and the final beliefs are the marginals of $\tilde{P}_\Phi$. As we now show, these two properties coincide for any calibrated clique tree.

Theorem 10.4

Let $\mathcal{T}$ be a clique tree over Φ, and let $\beta_i(C_i)$ be a set of calibrated potentials for $\mathcal{T}$. Then, $\tilde{P}_\Phi(\mathcal{X}) \propto Q_{\mathcal{T}}$ if and only if, for each $i \in \mathcal{V}_{\mathcal{T}}$, we have that $\beta_i(C_i) \propto \tilde{P}_\Phi(C_i)$.

Proof Let r be any clique in $\mathcal{T}$, which we choose to be the root. Define the descendant cliques of a clique C_i to be the cliques that are downstream from C_i relative to C_r; the nondescendant cliques are then the remaining cliques (other than C_i). Let X be the variables in the scope of the nondescendant cliques. It follows immediately from theorem 10.2 that

$$\tilde{P}_\Phi \models (C_i \perp X \mid S_{i,p_r(i)}).$$

From this, we obtain, using the standard chain-rule argument, that:

$$\tilde{P}_\Phi(\mathcal{X}) = \tilde{P}_\Phi(C_r) \cdot \prod_{i \neq r} \tilde{P}_\Phi(C_i \mid S_{i,p_r(i)}).$$

We can rewrite equation (10.11) as a similar product, using the same root:

$$Q_{\mathcal{T}}(\mathcal{X}) = \beta_r(C_r) \cdot \prod_{i \neq r} \beta_i(C_i \mid S_{i,p_r(i)}).$$

The "if" direction now follows from direct substitution of β_i for each $\tilde{P}_\Phi(\boldsymbol{C}_i)$.

To prove the "only if" direction, we note that each of the terms $\beta_i(\boldsymbol{C}_i \mid \boldsymbol{S}_{i,p_r(i)})$ is a conditional distribution; hence, if we marginalize out the variables not in $\boldsymbol{C}_r$ in the distribution $Q_\mathcal{T}$, each of these conditional distributions marginalizes to 1, and so we are left with $Q_\mathcal{T}(\boldsymbol{C}_r) = \beta_r(\boldsymbol{C}_r)$. It now follows that if $\tilde{P}_\Phi \propto Q_\mathcal{T}$, then $\tilde{P}_\Phi(\boldsymbol{C}_r) \propto Q_\mathcal{T}(\boldsymbol{C}_r) = \beta_r(\boldsymbol{C}_r)$. Because this argument applies to any choice of root clique, we have proved that this equality holds for every clique. ■

☞ **Thus, we can view the clique tree as an alternative representation of the joint measure, one that directly reveals the clique marginals.** As we will see, this view turns out to be very useful, both in the next section and in chapter 11.

10.3 Message Passing: Belief Update

The previous section showed one approach to message passing in clique trees, based on the same ideas of variable elimination that we discussed in chapter 9. In this section, we present a related approach, but one that is based on very different intuitions. We begin by describing an alternative message passing scheme that is different from but mathematically equivalent to that of the previous section. We then show how this new approach can be viewed as operations on the reparameterization of the distribution in terms of the clique and sepset beliefs $\{\beta_i(\boldsymbol{C}_i)\}_{i \in \mathcal{V}_\mathcal{T}}$ and $\{\mu_{i,j}(\boldsymbol{S}_{i,j})\}_{(i-j) \in \mathcal{E}_\mathcal{T}}$. Each message passing step will change this representation while leaving it a reparameterization of $\tilde{P}_\Phi$.

10.3.1 Message Passing with Division

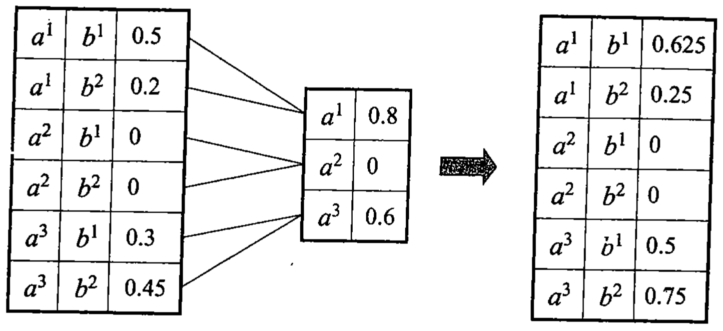
Consider again the message passing process used in CTree-SP-Calibrate (algorithm 10.2). There, two messages are passed along each link $(i–j)$. Assume, without loss of generality, that the first message is passed from $\boldsymbol{C}_j$ to $\boldsymbol{C}_i$. A return message from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$ is passed when $\boldsymbol{C}_i$ has received messages from all of its other neighbors.

At this point, $\boldsymbol{C}_i$ has all of the necessary information to compute its final potential. It multiplies the initial potential with the incoming messages from all of its neighbors:

$$\beta_i = \psi_i \cdot \prod_{k \in \mathrm{Nb}_i} \delta_{k \to i}. \tag{10.12}$$

As we discussed, this final potential is not used in computing the message to $\boldsymbol{C}_j$: this potential already incorporates the information (message) passed from $\boldsymbol{C}_j$; if we used it when computing the message to $\boldsymbol{C}_j$, this information would be double-counted. Thus, the message from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$ is computed in a way that omits the information obtained from $\boldsymbol{C}_j$: we multiply the initial potential with all of the messages except for the message from $\boldsymbol{C}_i$, and then marginalize over the sepset (equation (10.2)).

A different approach to computing the same expression is to multiply in *all* of the messages, and then *divide* the resulting factor by $\delta_{j \to i}$. To make this notion precise, we must define a factor-division operation:



a^1	b^1	0.5
a^1	b^2	0.2
a^2	b^1	0
a^2	b^2	0
a^3	b^1	0.3
a^3	b^2	0.45

a^1	0.8
a^2	0
a^3	0.6

a^1	b^1	0.625
a^1	b^2	0.25
a^2	b^1	0
a^2	b^2	0
a^3	b^1	0.5
a^3	b^2	0.75

Figure 10.7 An example of factor division

Definition 10.7
factor division

Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be disjoint sets of variables, and let $\phi_1(\boldsymbol{X}, \boldsymbol{Y})$ and $\phi_2(\boldsymbol{Y})$ be two factors. We define the division $\frac{\phi_1}{\phi_2}$ *to be a factor ψ of scope $\boldsymbol{X}, \boldsymbol{Y}$ defined as follows:*

$$\psi(\boldsymbol{X}, \boldsymbol{Y}) = \frac{\phi_1(\boldsymbol{X}, \boldsymbol{Y})}{\phi_2(\boldsymbol{Y})},$$

where we define $0/0 = 0$. ■

Note that, as in the case of other factor operations, factor division is done component by component. Figure 10.7 shows an example. Also note that the operation is not well defined if the denominator is zero and the numerator is not.

We now see that we can compute the expression of equation (10.2) by computing the beliefs as in equation (10.12), and then dividing by the remaining message:

$$\delta_{i \to j} = \frac{\sum_{C_i - S_{i,j}} \beta_i}{\delta_{j \to i}}. \tag{10.13}$$

Example 10.7

Let us return to the simple clique tree in example 10.5, and assume that C_2 serves as the (de facto) root, so that we first pass messages from C_1 to C_2 and from C_3 to C_2. The message $\delta_{1 \to 2}$ is computed as $\sum_A \psi_1(A, B)$. Using the variable elimination message (CTree-SP-Calibrate)*, we pass a return message $\delta_{2 \to 1}(B) = \sum_C \psi_2(B, C)\delta_{3 \to 2}(C)$. Alternatively, we can compute $\beta_2(B, C) = \delta_{1 \to 2}(B) \cdot \delta_{3 \to 2}(C) \cdot \psi_2(B, C)$, and then send a message*

$$\frac{\sum_C \beta_2(B, C)}{\delta_{1 \to 2}(B)} = \sum_C \frac{\beta_2(B, C)}{\delta_{1 \to 2}(B)} = \sum_C \psi_2(B, C) \cdot \delta_{3 \to 2}(C).$$

Thus, the two approaches are equivalent. ■

sum-product-divide
beliefs

Based on this insight, we can define the *sum-product-divide* message passing scheme, where each clique C_i maintains its fully updated current beliefs β_i, which are defined as in equation (10.8). Each sepset also maintains its beliefs $\mu_{i,j}$ defined as the product of the messages in both directions, as in equation (10.9). We now show that the entire message passing process can be executed in an equivalent way in terms of the clique and sepset beliefs, without having to remember the initial potentials ψ_i or to compute explicitly the messages $\delta_{i \to j}$.

The message passing process follows the lines of example 10.7. Each clique C_i initializes β_i as ψ_i and then updates it by multiplying with message updates received from its neighbors. Each sepset $S_{i,j}$ maintains $\mu_{i,j}$ as the previous message passed along the edge $(i–j)$, regardless of the direction. This message is used to ensure that we do not double-count: Whenever a new message is passed along the edge, it is divided by the old message, eliminating the previous message from the update to the clique. Somewhat surprisingly, as we will show, the message passing operation is correct regardless of the clique that sent the last message on the edge. Intuitively, once the message is passed, its information is incorporated into both cliques; thus, each needs to divide by it when passing a message to the other. We can view this algorithm as maintaining a set of belief over the cliques in the tree. The message passing operation takes the beliefs of one clique and uses them to update the beliefs of a neighbor. Thus, we call this algorithm *belief-update* message passing; it is also known as the *Lauritzen-Spiegelhalter algorithm.*

belief-update

Example 10.8

Continuing with example 10.7, assume that C_2 initially passes an uninformed message to C_3: $\sigma_{2 \to 3} = \sum_B \psi_2(B, C)$. This message multiplies the beliefs about C_3, so that, at this point:

$$\beta_3(C, D) = \psi_3(C, D) \sum_B \psi_2(B, C).$$

This message is also stored in the sepset as $\mu_{2,3}$. Now, assume that C_3 sends a message to C_2: $\sigma_{3 \to 2}(C) = \sum_D \beta_3(C, D)$. This message is divided by $\mu_{2,3}$, so the actual update for C_2 is:

$$\begin{aligned} \frac{\sigma_{3 \to 2}(C)}{\mu_{2,3}(C)} &= \frac{\sum_D \beta_3(C, D)}{\mu_{2,3}(C)} \\ &= \frac{\sum_D \psi_3(C, D)\mu_{2,3}(C)}{\mu_{2,3}(C)} \\ &= \sum_D \psi_3(C, D). \end{aligned}$$

This expression is precisely the update that C_2 would have received from C_3 in the case where C_2 does not first send an uninformed message. At this point, the message stored in the sepset is

$$\sum_D \beta_3(C, D) = \sum_D \left(\psi_3(C, D) \cdot \sum_B \psi_2(B, C) \right).$$

Assume that at the next step C_2 receives a message from C_1, containing $\sum_A \psi_1(A, B)$. The sepset $S_{1,2}$ contains a message that is identically 1, so that this message is transmitted as is. At this point, C_2 has received informed messages from both sides and is therefore informed. Indeed, we have shown that:

$$\beta_2(B, C) = \psi_2(B, C) \cdot \sum_A \psi_1(A, B) \cdot \sum_D \psi_3(C, D).$$

as required. ■

Algorithm 10.3 Calibration using belief propagation in clique tree

```
   Procedure CTree-BU-Calibrate (
     $\Phi$,   // Set of factors
     $\mathcal{T}$   // Clique tree over $\Phi$
   )
1    Initialize-CTree
2    while exists an uninformed clique in $\mathcal{T}$
3      Select $(i\text{–}j) \in \mathcal{E}_{\mathcal{T}}$
4      BU-Message$(i, j)$
5    return $\{\beta_i\}$

   Procedure Initialize-CTree (
   )
1    for each clique $C_i$
2      $\beta_i \leftarrow \prod_{\phi \,:\, \alpha(\phi)=i} \phi$
3    for each edge $(i\text{–}j) \in \mathcal{E}_{\mathcal{T}}$
4      $\mu_{i,j} \leftarrow \mathbf{1}$

   Procedure BU-Message (
     i,   // sending clique
     j    // receiving clique
   )
1    $\sigma_{i \rightarrow j} \leftarrow \sum_{C_i - S_{i,j}} \beta_i$
2      // marginalize the clique over the sepset
3    $\beta_j \leftarrow \beta_j \cdot \frac{\sigma_{i \rightarrow j}}{\mu_{i,j}}$
4    $\mu_{i,j} \leftarrow \sigma_{i \rightarrow j}$
```

The precise algorithm is shown in algorithm 10.3. Note that, as written, the message passing algorithm is underspecified: in line 3, we can select any pair of cliques C_i and C_j between which we will pass a message. Interestingly, we can make this choice arbitrarily, without damaging the correctness of the algorithm. For example, if C_i (for some reason) passes the same message to C_j a second time, the process of dividing out by the stored message reduces the message actually passed to $\mathbf{1}$, so that it has no influence. Furthermore, if C_i passes a message to C_j based on partial information (that is, without taking into consideration all of its incoming messages), and then resends a more updated message later on, the effect is identical to simply sending the updated message once. Moreover, at convergence, regardless of the message passing steps used, we necessarily have a calibrated clique tree. This property follows from the fact that, in order for all message updates to have no effect, we need to have

$\sigma_{i\to j} = \mu_{i,j} = \sigma_{j\to i}$ for all i, j, and so:

$$\sum_{C_i - S_{i,j}} \beta_i = \mu_{i,j} = \sum_{C_j - S_{i,j}} \beta_j.$$

Thus, at convergence, each pair of neighboring cliques i, j must agree on the variables in sepset, and the message $\mu_{i,j}$ is precisely the sepset marginal. These properties also follow from the equivalence between belief-update message passing and sum-product message passing, which we show next.

10.3.2 Equivalence of Sum-Product and Belief Update Messages

So far, although we used sum-product message propagation to motivate the definition of the belief update steps, we have not shown a direct connection between them. We now show a simple and elegant equivalence between the two types of message passing operations. From this result, it immediately follows that belief-update message passing is guaranteed to converge to the correct marginals.

Our proof is based on equation (10.8) and equation (10.9), which provide a mapping between the sum-product and belief-update representations. We consider corresponding runs of the two algorithms in which an identical sequence of message passing steps is executed. We show that these two properties hold as an invariant between the data structures maintained by the two algorithms. The invariant holds initially, and it is maintained throughout the corresponding runs.

Theorem 10.5 *Consider a set of sum-product initial potentials $\{\psi_i : i \in \mathcal{V}_\mathcal{T}\}$ and messages $\{\delta_{i\to j}, \delta_{j\to i} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}\}$, and a set of belief-update beliefs $\{\beta_i : i \in \mathcal{V}_\mathcal{T}\}$ and messages $\{\mu_{i,j} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}\}$, for which equation (10.8) and equation (10.9) hold. For any pair of neighboring cliques C_k, C_l, let $\{\delta'_{i\to j}, \delta'_{j\to i} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}\}$ be the set of sum-product messages following an application of* SP-Message(i, j), *and $\{\beta'_i : C_i \in \mathcal{T}\}, \{\mu'_{i,j} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}\}$, be the set of belief-update beliefs following an application of* BU-Message(i, j). *Then equation (10.8) and equation (10.9) also hold for the new beliefs $\delta'_{i\to j}, \beta'_i, \mu'_{i,j}$.*

The proof uses simple algebraic manipulation, and it is left as an exercise (exercise 10.4).

This equivalence implies another result that will prove important in subsequent developments:

Corollary 10.3 *In an execution of belief-update message passing, the clique tree invariant equation (10.10) holds initially and after every message passing step.*

Proof The proof of proposition 10.3 relied only on equation (10.8) and equation (10.9). Because these two equalities hold in every step of the belief-update message passing algorithm, we have that the clique tree invariant also holds continuously. ■

This equivalence also allows us to define a message schedule that guarantees convergence to the correct clique marginals in two passes: We simply follow the same upward-downward-pass schedule used in CTree-SP-Calibrate, using any (arbitrarily chosen) root clique C_r.

10.3.3 Answering Queries

As we have seen, a calibrated clique tree contains the answer to multiple queries at once: the posterior probability of any set of variables that are present together in a single clique. A particular type of query that turns out to be important in this setting is the computation of the posterior for families of variables in a probabilistic network: a node and its parents in the context of Bayesian networks, or a clique in a Markov network. The family preservation property for cluster graphs (and hence for clique trees) implies that a family must be a subset of some cluster in the cluster graph.

In addition to these queries, which we get immediately as a by-product of calibration, we can also use a clique tree for other queries. We describe the algorithm for these queries in terms of a calibrated clique tree that satisfies the clique tree invariant. Due to the equivalence of sum-product and belief-update message passing, we can obtain such a clique tree using either method.

10.3.3.1 Incremental Updates

Consider a situation where, at some point in time, we have a certain set of observations, which we use to condition our distribution and reach conclusions. At some later time, we obtain additional evidence, and want to update our conclusions accordingly. This type of situation, where we want to perform *incremental update* is very common in a wide variety of settings. For example, in a medical setting, we often perform diagnosis on the basis of limited evidence; the initial diagnosis helps us decide which tests to perform, and the results need to be incorporated into our diagnosis.

incremental update

The most naive approach to dealing with this task is simply to condition the initial factors (for example, the CPDs) on all of the evidence, and then redo the calibration process from the beginning, starting from these factors. A somewhat more efficient approach is based on the view of the clique tree as representing the distribution $\tilde{P}_\Phi$.

Assume that our initial distribution $\tilde{P}_\Phi$ (prior to the new information) is represented via a set of factors Φ, as in equation (10.1). Given some evidence $Z = z$, we can obtain $\tilde{P}_\Phi(\mathcal{X}, Z = z)$ by zeroing out the entries in the unnormalized distribution that are inconsistent with the evidence $Z = z$. We can accomplish this effect by multiplying $\tilde{P}_\Phi$ with an additional factor which is the indicator function $\boldsymbol{1}\{Z = z\}$. More precisely, assume that our current distribution over $\mathcal{X}$ is defined by a set of factors Φ, so that

$$\tilde{P}_\Phi(\mathcal{X}) = \prod_{\phi \in \Phi} \phi.$$

Then,

$$\tilde{P}_\Phi(\mathcal{X}, Z = z) = \boldsymbol{1}\{Z = z\} \cdot \prod_{\phi \in \Phi} \phi.$$

Let $\tilde{P}'_\Phi(\mathcal{X}) = \tilde{P}_\Phi(\mathcal{X}, Z = z)$.

Now, assume that we have a clique tree (calibrated or not) that represents this distribution using the clique tree invariant. That is:

$$\tilde{P}_\Phi(\mathcal{X}) = Q_\mathcal{T} = \frac{\prod_{i \in \mathcal{V}_\mathcal{T}} \beta_i(\boldsymbol{C}_i)}{\prod_{(i-j) \in \mathcal{E}_\mathcal{T}} \mu_{i,j}(\boldsymbol{S}_{i,j})}.$$

We can represent the distribution $\tilde{P}'_\Phi(\mathcal{X})$ as

$$\tilde{P}'_\Phi(\mathcal{X}) = \boldsymbol{1}\{Z = z\} \cdot \frac{\prod_{i \in \mathcal{V}_\mathcal{T}} \beta_i(\boldsymbol{C}_i)}{\prod_{(i-j) \in \mathcal{E}_\mathcal{T}} \mu_{i,j}(\boldsymbol{S}_{i,j})}.$$

Thus, we obtain a representation of $\tilde{P}'_\Phi$ in the clique tree simply by multiplying in the new factor $\boldsymbol{1}\{Z = z\}$ into some clique $\boldsymbol{C}_i$ containing the variable Z.

If the clique tree is calibrated before this new factor is introduced, then the clique $\boldsymbol{C}_i$ has already assimilated all of the other information in the graph. Thus, the clique $\boldsymbol{C}_i$ itself is now fully informed, and no additional message passing is required in order to obtain $\tilde{P}'_\Phi(\boldsymbol{C}_i)$. Other cliques, however, still need to be updated with the new information. To obtain $\tilde{P}'_\Phi(\boldsymbol{C}_j)$ for another clique $\boldsymbol{C}_j$, we need only transmit messages from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$, via the intervening cliques on the path between them. (See exercise 10.10.) As a consequence, the entire tree can be recalibrated to account for the new evidence using a single pass. Note that retracting evidence is not as simple: Once we multiply parts of the distribution by zero, these parts are lost, and they cannot be recovered. Thus, if we want to reserve the ability to retract evidence, we must store the beliefs prior to the conditioning step (see exercise 10.12).

Interestingly, the same incremental-update approach applies to other forms of updating the distribution. In particular, we can multiply the distribution with a factor that is not an indicator function for some variable, an operation that is useful in various applications. The same analysis holds unchanged.

10.3.3.2 Queries Outside a Clique

Consider a query $P(\boldsymbol{Y} \mid e)$ where the variables $\boldsymbol{Y}$ are not present together in a single clique. One naive approach is to construct a clique tree where we force one of the cliques to contain $\boldsymbol{Y}$ (see exercise 10.13). However, this approach forces us to tailor our clique tree to different queries, negating many of its advantages. An alternative approach is to perform variable elimination over a calibrated clique tree.

Example 10.9

Consider the simple clique tree of example 10.7, and assume that we have calibrated the clique tree, so that the beliefs represent the joint distribution as in equation (10.10). Assume that we now want to compute the probability $\tilde{P}_\Phi(B, D)$. If the entire clique tree is calibrated, so is any (connected) subtree $\mathcal{T}'$. Letting $\mathcal{T}'$ consist of the two cliques $\boldsymbol{C}_2$ and $\boldsymbol{C}_3$, it follows from theorem 10.4 that:

$$\tilde{P}_\Phi(B, C, D) = Q_{\mathcal{T}'}.$$

By the clique tree invariant (equation (10.10)), we have that:

$$\begin{aligned}
\tilde{P}_\Phi(B, D) &= \sum_C \tilde{P}_\Phi(B, C, D) \\
&= \sum_C \frac{\beta_2(B, C)\beta_3(C, D)}{\mu_{2,3}(C)} \\
&= \sum_C \tilde{P}_\Phi(B \mid C)\tilde{P}_\Phi(C, D),
\end{aligned}$$

where the last equality follows from calibration. Each of these probability expressions corresponds to a set of clique beliefs divided by a message. We can now perform variable elimination, using these factors in the usual way. ■

Algorithm 10.4 Out-of-clique inference in clique tree

```
Procedure CTree-Query (
    $\mathcal{T}$,   // Clique tree over $\Phi$
    $\{\beta_i\}, \{\mu_{i,j}\}$,   // Calibrated clique and sepset beliefs for $\mathcal{T}$
    $\boldsymbol{Y}$   // A query
  )
1   Let $\mathcal{T}'$ be a subtree of $\mathcal{T}$ such that $\boldsymbol{Y} \subseteq Scope[\mathcal{T}']$
2   Select a clique $r \in \mathcal{V}_{\mathcal{T}'}$ to be the root
3   $\Phi \leftarrow \beta_r$
4   for each $i \in \mathcal{V}'_{\mathcal{T}}$
5     $\phi \leftarrow \frac{\beta_i}{\mu_{i,p_r(i)}}$
6     $\Phi \leftarrow \Phi \cup \{\phi\}$
7   $\boldsymbol{Z} \leftarrow Scope[\mathcal{T}'] - \boldsymbol{Y}$
8   Let $\prec$ be some ordering over $\boldsymbol{Z}$
9   return Sum-Product-Variable-Elimination($\Phi, \boldsymbol{Z}, \prec$)
```

More generally, we can compute the joint probability $\tilde{P}_\Phi(\boldsymbol{Y})$ for an arbitrary subset $\boldsymbol{Y}$ by using the beliefs in a calibrated clique tree to define factors corresponding to conditional probabilities in $\tilde{P}_\Phi$, and then performing variable elimination over the resulting set of factors. The precise algorithm is shown in algorithm 10.4. The savings over simple variable elimination arise because we do not have to perform inference over the entire clique tree, but only over a portion of the tree that contains the variables $\boldsymbol{Y}$ that constitute our query. In cases where we have a very large clique tree, the savings can be significant.

10.3.3.3 Multiple Queries

Now, assume that we want to compute the probabilities of an entire set of queries where the variables are not together in a clique. For example, we might wish to compute $\tilde{P}_\Phi(X, Y)$ for every pair of variables $X, Y \in \mathcal{X} - \boldsymbol{E}$. Clearly, the approach of constructing a clique tree to ensure that our query variables are present in a single clique breaks down in this case: If every pair of variables is present in some clique, there must be some clique that contains all of the variables (see exercise 10.14).

A somewhat less naive approach is simply to run the variable elimination algorithm of algorithm 10.4 $\binom{n}{2}$ times, once for each pair of variables X, Y. However, because pairs of variables, on average, are fairly far from each other in the clique tree, this approach requires fairly substantial running time (see exercise 10.15). An even better approach can be obtained by using *dynamic programming*.

dynamic programming

Consider a calibrated clique tree $\mathcal{T}$ over Φ, and assume we want to compute the probability $\tilde{P}_\Phi(X, Y)$ for every pair of variables X, Y. We execute this process by gradually constructing a table for each C_i, C_j that contains $\tilde{P}_\Phi(C_i, C_j)$. We construct the table for i, j in order of the distance between C_i and C_j in the tree.

The base case is when i, j are neighboring cliques. In this case, we simply extract $\tilde{P}_\Phi(C_i)$ from its clique beliefs, and compute

$$\tilde{P}_\Phi(C_j \mid C_i) = \frac{\beta_j(C_j)}{\mu_{i,j}(C_i \cap C_j)}.$$

From these, we can compute $\tilde{P}_\Phi(C_i, C_j)$.

Now, consider a pair of cliques C_i, C_j that are not neighbors, and let C_l be the neighbor of C_j that is one step closer in the clique tree to C_i. By construction, we have already computed $\tilde{P}_\Phi(C_i, C_l)$ and $\tilde{P}_\Phi(C_l, C_j)$. The key now, is to observe that

$$\tilde{P}_\Phi \models (C_i \perp C_j \mid C_l).$$

Thus, we can compute

$$\tilde{P}_\Phi(C_i, C_j) = \sum_{C_l - C_j} \tilde{P}_\Phi(C_i, C_l)\tilde{P}_\Phi(C_j \mid C_l),$$

where $\tilde{P}_\Phi(C_j \mid C_l)$ can be easily computed from the marginal $\tilde{P}_\Phi(C_j, C_l)$.

The cost of this computation is significantly lower than that of running variable elimination in the clique tree $\binom{n}{2}$ times (see exercise 10.15).

10.4 Constructing a Clique Tree

So far, we have assumed that a clique tree is given to us. How do we construct a clique tree for a set of factors, or, equivalently, for its underlying undirected graph $\mathcal{H}_\Phi$? There are two basic approaches, the first based on variable elimination and the second on direct graph manipulation.

10.4.1 Clique Trees from Variable Elimination

The first approach is based on variable elimination. As we discussed in section 10.1.1, the execution of a variable elimination algorithm can be associated with a cluster graph: A cluster C_i corresponds to the factor ψ_i generated during the execution of the algorithm, and an undirected edge connects C_i and C_j when τ_i is used (directly) in the computation of ψ_j (or vice versa). As we showed in section 10.1.1, this cluster graph is a tree, and it satisfies the running intersection property; hence, it is a clique tree.

As we showed in theorem 9.6, each factor in an execution of variable elimination with the ordering $\prec$ is a subset of a clique in the induced graph $\mathcal{I}_{\Phi,\prec}$. Furthermore, every maximal clique is a factor in the computation. Based on this result, we can conclude that, in the clique tree $\mathcal{T}$ induced by variable elimination using the ordering $\prec$, each clique is also a clique in the induced graph $\mathcal{I}_{\Phi,\prec}$, and each clique in $\mathcal{I}_{\Phi,\prec}$ is a clique in $\mathcal{T}$. This equivalence is the reason for the use of term *clique* in this context.

In the context of clique tree inference, it is standard to reduce the tree to contain only clusters that are (maximal) cliques in $\mathcal{I}_{\Phi,\prec}$. Specifically, we eliminate from the tree a cluster C_j which is a strict subset of some other cluster C_i: