parameters. In fact, for the case of linear exponential families, we want to strengthen our desiderata, and require that any parameter vector in $\mathcal{R}^K$ defines a distribution in the family. Unfortunately, as stated, this desideratum is not always achievable. To understand why, recall that the definition of a legal parameter space Θ requires that each parameter vector $\boldsymbol{\theta} \in \Theta$ give rise to a legal (normalizable) distribution $P_{\boldsymbol{\theta}}$. These normalization requirements can impose constraints on the space of legal parameters.

Example 8.4

Consider again the Gaussian distribution. Suppose we define a new parameter space using the definition of t. *That is let* $\boldsymbol{\eta} = \mathtt{t}(\mu, \sigma^2) = \langle \frac{2\mu}{2\sigma^2}, -\frac{1}{2\sigma^2} \rangle$ *be the natural parameters that corresponds to* $\boldsymbol{\theta} = \langle \mu, \sigma^2 \rangle$. *Clearly, we can now write*

$$P_{\boldsymbol{\eta}}(x) \propto \exp\{\langle \eta, \tau(x) \rangle\}.$$

However, not every choice of $\boldsymbol{\eta}$ *would lead to a legal distribution. For the distribution to be normalized, we need to be able to compute*

$$\begin{aligned} Z(\boldsymbol{\eta}) &= \int \exp\{\langle \boldsymbol{\eta}, \tau(x) \rangle\}\, dx \\ &= \int_{-\infty}^{\infty} \exp\left\{\eta_1 x + \eta_2 x^2\right\} dx. \end{aligned}$$

If $\eta_2 \geq 0$ *this integral is undefined, since the function grows when* x *approaches* ∞ *and* $-\infty$. *When* $\eta_2 < 0$, *the integral has a finite value. Fortunately, if we consider* $\boldsymbol{\eta} = \mathtt{t}(\mu, \sigma^2)$ *of equation (8.5), we see that the second component is always negative (since* $\sigma^2 > 0$*). In fact, we can see that the* image *of the original parameter space,* $\langle \mu, \sigma^2 \rangle \in \mathcal{R} \times \mathcal{R}^+$, *through the function* $\mathtt{t}(\mu, \sigma^2)$, *is the space* $\mathcal{R} \times \mathcal{R}^-$. *We can verify that, for every* $\boldsymbol{\eta}$ *in that space, the normalization constant is well defined.* ■

natural parameter space

More generally, when we consider natural parameters for a sufficient statistics function τ, we define the set of allowable natural parameters, the *natural parameter space*, to be the set of natural parameters that can be normalized

$$\Theta = \left\{ \boldsymbol{\theta} \in \mathcal{R}^K : \int \exp\{\langle \boldsymbol{\theta}, \tau(\xi) \rangle\}\, d\xi < \infty \right\}.$$

In the case of distributions over finite discrete spaces, all parameter choices lead to normalizable distributions, and so $\Theta = \mathcal{R}^K$. In other examples, such as the Gaussian distribution, the natural parameter space can be more constrained. An exponential family over the natural parameter space, and for which the natural parameter space is open and convex, is called a *linear exponential family*.

linear exponential family

The use of linear exponential families significantly simplifies the definition of a family. To specify such a family, we need to define only the function τ; all other parts of the definition are implicit based on this function. This gives us a tool to describe distributions in a concise manner. As we will see, linear exponential families have several additional attractive properties.

Where do find linear exponential families? The two examples we presented earlier were not phrased as linear exponential families. However, as we saw in example 8.4, we may be able to provide an alternative parameterization of a nonlinear exponential family as a linear exponential family. This example may give rise to the impression that any family can be reparameterized in a trivial manner. However, there are more subtle situations.

Example 8.5

Consider the Bernoulli distribution. Again, we might reparameterize θ by $\mathtt{t}(\theta)$. However, the image of the function $\mathtt{t}$ of example 8.2 is the curve $\langle \ln\theta, \ln(1-\theta)\rangle$. This curve is not a convex set, and it is clearly a subspace of the natural parameter space.

Alternatively, we might consider using the entire natural parameter space $\mathcal{R}^2$, corresponding to the sufficient statistic function $\tau(X) = \langle \mathbf{1}\{X = x^1\}, \mathbf{1}\{X = x^0\}\rangle$ of equation (8.2). This gives rise to the parametric form:

$$P_{\boldsymbol{\theta}}(x) \propto \exp\{\langle \boldsymbol{\theta}, \tau(x)\rangle\} = \exp\left\{\theta_1 \mathbf{1}\{X = x^1\} + \theta_2 \mathbf{1}\{X = x^0\}\right\}.$$

Because the probability space is finite, this form does define a distribution for every choice of $\langle\theta_1, \theta_2\rangle$. However, it is not difficult to verify that this family is redundant: for every constant c, the parameters $\langle\theta_1 + c, \theta_2 + c\rangle$ define the same distribution as $\langle\theta_1, \theta_2\rangle$.

Thus, a two-dimensional space is overparameterized for this distribution; conversely, the one-dimensional subspace defined by the natural parameter function is not well behaved. The solution is to use an alternative representation of a one-dimensional space. Since we have a redundancy, we may as well clamp θ_2 to be 0. This results in the following representation of the Bernoulli distribution:

$$\begin{aligned} \tau(x) &= \mathbf{1}\{x = x^1\} \\ \mathtt{t}(\theta) &= \ln\frac{\theta}{1-\theta}. \end{aligned}$$

We see that

$$\begin{aligned} \exp\left\{\langle \mathtt{t}(\theta), \tau(x^1)\rangle\right\} &= \frac{\theta}{1-\theta} \\ \exp\left\{\langle \mathtt{t}(\theta), \tau(x^0)\rangle\right\} &= 1. \end{aligned}$$

Thus,

$$Z(\theta) = 1 + \frac{\theta}{1-\theta} = \frac{1}{1-\theta}.$$

Using these, we can verify that

$$P_\theta(x^1) = (1-\theta)\frac{\theta}{1-\theta} = \theta.$$

We conclude that this exponential representation captures the Bernoulli distribution. Notice now that, in the new representation, the image of $\mathtt{t}$ is the whole real line $\mathcal{R}$. Thus, we can define a linear exponential family with this sufficient statistic function. ■

Example 8.6

Now, consider a multinomial variable X with k values $x^1, \ldots, x^k$. The situation here is similar to the one we had with the Bernoulli distribution. If we use the simplest exponential representation, we find that the legal natural parameters are on a curved manifold of $\mathcal{R}^k$. Thus, instead we define the sufficient statistic as a function from values of x to $\mathcal{R}^{k-1}$:

$$\tau(x) = \langle \mathbf{1}\{x = x^2\}, \ldots, \mathbf{1}\{x = x^k\}\rangle.$$

Using a similar argument as with the Bernoulli distribution, we see that if we define

$$\mathtt{t}(\boldsymbol{\theta}) = \langle \ln \frac{\theta_2}{\theta_1}, \ldots, \ln \frac{\theta_k}{\theta_1} \rangle,$$

then we reconstruct the original multinomial distribution. It is also easy to check that the image of $\mathtt{t}$ *is* $\mathcal{R}^{k-1}$*. Thus, by reparameterizing, we get a linear exponential family.* ■

All these examples define linear exponential families. An immediate question is whether there exist families that are not linear. As we will see, there are such cases. However, the examples we present require additional machinery.

8.3 Factored Exponential Families

The two examples of exponential families so far were of univariate distributions. Clearly, we can extend the notion to multivariate distributions as well. In fact, we have already seen one such example. Recall that, in definition 4.15, we defined log-linear models as distributions of the form:

$$P(X_1, \ldots, X_n) \propto \exp\left\{\sum_{i=1}^{k} \theta_i \cdot f_i(\boldsymbol{D}_i)\right\}$$

where each feature f_i is a function whose scope is $\boldsymbol{D}_i$. Such a distribution is clearly a linear exponential family where the sufficient statistics are the vector of features

$$\tau(\xi) = \langle f_1(\boldsymbol{d}_1), \ldots, f_k(\boldsymbol{d}_k) \rangle.$$

As we have shown, by choosing the appropriate features, we can devise a log-linear model to represent a given discrete Markov network structure. This suffices to show that discrete Markov networks are linear exponential families.

8.3.1 Product Distributions

What about other distributions with product forms? Initially the issues seem deceptively easy. A product form of terms corresponds to a simple composition of exponential families

Definition 8.2 exponential factor family

An (unnormalized) exponential factor family Φ *is defined by* τ, $\mathtt{t}$, A, *and* Θ *(as in the exponential family). A factor in this family is*

$$\phi_{\boldsymbol{\theta}}(\xi) = A(\xi) \exp\left\{\langle \mathtt{t}(\boldsymbol{\theta}), \tau(\xi) \rangle\right\}.$$

■

Definition 8.3 family composition

Let $\Phi_1, \ldots, \Phi_k$ *be exponential factor families, where each* Φ_i *is specified by* τ_i, $\mathtt{t}_i$, A_i, *and* Θ_i*. The* composition *of* $\Phi_1, \ldots, \Phi_k$ *is the family* $\Phi_1 \times \Phi_2 \times \cdots \times \Phi_k$ *parameterized by* $\boldsymbol{\theta} = \boldsymbol{\theta}_1 \circ \boldsymbol{\theta}_2 \circ \cdots \circ \boldsymbol{\theta}_k \in \Theta_1 \times \Theta_2 \times \cdots \times \Theta_k$*, defined as*

$$P_{\boldsymbol{\theta}}(\xi) \propto \prod_i \phi_{\boldsymbol{\theta}_i}(\xi) = \left(\prod_i A_i(\xi)\right) \exp\left\{\sum_i \langle \mathtt{t}_i(\boldsymbol{\theta}_i), \tau_i(\xi) \rangle\right\}$$

where $\phi_{\boldsymbol{\theta}_i}$ *is a factor in the* i*'th factor family.* ■

It is clear from this definition that the composition of exponential factors is an exponential family with $\tau(\xi) = \tau_1(\xi) \circ \tau_2(\xi) \circ \cdots \circ \tau_k(\xi)$ and natural parameters $\mathtt{t}(\boldsymbol{\theta}) = \mathtt{t}_1(\boldsymbol{\theta}_1) \circ \mathtt{t}_2(\boldsymbol{\theta}_2) \circ \cdots \circ \mathtt{t}_k(\boldsymbol{\theta}_k)$.

This simple observation suffices to show that if we have exponential representation for potentials in a Markov network (not necessarily simple potentials), then their product is also an exponential family. Moreover, it follows that the product of linear exponential factor families is a linear exponential family.

8.3.2 Bayesian Networks

Taking the same line of reasoning, we can also show that, if we have a set of CPDs from an exponential family, then their product is also in the exponential family. Thus, we can conclude that a Bayesian network with exponential CPDs defines an exponential family. To show this, we first note that many of the CPDs we saw in previous chapters can be represented as exponential factors.

Example 8.7

We start by examining a simple table-CPD $P(X \mid \boldsymbol{U})$. Similar to the case of Bernoulli distribution, we can define the sufficient statistics to be indicators for different entries in $P(X \mid \boldsymbol{U})$. Thus, we set

$$\tau_{P(X|\boldsymbol{U})}(\mathcal{X}) = \langle \boldsymbol{I}\{X = x, \boldsymbol{U} = \boldsymbol{u}\} : x \in Val(X), \boldsymbol{u} \in Val(\boldsymbol{U}) \rangle.$$

We set the natural parameters to be the corresponding parameters

$$\mathtt{t}_{P(X|\boldsymbol{U})}(\boldsymbol{\theta}) = \langle \ln P(x \mid \boldsymbol{u}) : x \in Val(X), \boldsymbol{u} \in Val(\boldsymbol{U}) \rangle.$$

It is easy to verify that

$$P(x \mid \boldsymbol{u}) = \exp\left\{ \langle \mathtt{t}_{P(X|\boldsymbol{U})}(\boldsymbol{\theta}), \tau_{P(X|\boldsymbol{U})}(x, \boldsymbol{u}) \rangle \right\},$$

since exactly one entry of $\tau_{P(X|\boldsymbol{U})}(x, \boldsymbol{u})$ is 1 and the rest are 0. Note that this representation is not a linear exponential factor. ■

Clearly, we can use the same representation to capture any CPD for discrete variables. In some cases, however, we can be more efficient. In tree-CPDs, for example, we can have a feature set for each leaf in tree, since all parent assignment that reach the leaf lead to the same parameter over the children.

What happens with continuous CPDs? In this case, not every CPD can be represented by an exponential factor. However, some cases can.

Example 8.8

Consider a linear Gaussian CPD for $P(X \mid \boldsymbol{U})$ where

$$X = \beta_0 + \beta_1 u_1 + \cdots + \beta_k u_k + \epsilon,$$

where ϵ is a Gaussian random variable with mean 0 and variance σ^2, representing the noise in the system. Stated differently, the conditional density function of X is

$$P(x \mid \boldsymbol{u}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{1}{2\sigma^2}(x - (\beta_0 + \beta_1 u_1 + \cdots + \beta_k u_k))^2 \right\}.$$

By expanding the squared term, we find that the sufficient statistics are the first and second moments of all the variables

$$\tau_{P(X|\boldsymbol{U})}(\mathcal{X}) = \langle 1, x, u_1, \ldots, u_k, x^2, xu_1, \ldots, xu_k, u_1^2, u_1u_2, \ldots, u_k^2 \rangle,$$

and the natural parameters are the coefficients of each of these terms. ■

As the product of exponential factors is an exponential family, we conclude that a Bayesian network that is the product of CPDs that have exponential form defines an exponential family.

However, there is one subtlety that arises in the case of Bayesian networks that does not arise for a general product form. When we defined the product of a set of exponential factors in definition 8.3, we ignored the partition functions of the individual factors, allowing the partition function of the overall distribution to ensure global normalization.

However, in both of our examples of exponential factors for CPDs, we were careful to construct a normalized conditional distribution. This allows us to use the chain rule to compose these factors into a joint distribution without the requirement of a partition function. This requirement turns out to be critical: We cannot construct a Bayesian network from a product of unnormalized exponential factors.

Example 8.9

Consider the network structure $A \rightarrow B$, with binary variables. Now, suppose we want to represent the CPD $P(B \mid A)$ using a more concise representation than the one of example 8.7. As suggested by example 8.5, we might consider defining

$$\tau(A, B) = \langle \mathbf{1}\{A = a^1\}, \mathbf{1}\{B = b^1, A = a^1\}, \mathbf{1}\{B = b^1, A = a^0\} \rangle.$$

That is, for each conditional distribution, we have an indicator only for one of the two relevant cases. The representation of example 8.5 suggests that we should define

$$\mathtt{t}(\boldsymbol{\theta}) = \left\langle \ln \frac{\theta_{a^1}}{\theta_{a^0}}, \ln \frac{\theta_{b^1|a^1}}{\theta_{b^0|a^1}}, \ln \frac{\theta_{b^1|a^0}}{\theta_{b^0|a^0}} \right\rangle.$$

Does this construction give us the desired distribution? Under this construction, we would have

$$P_{\boldsymbol{\theta}}(a^1, b^1) = \frac{1}{Z(\boldsymbol{\theta})} \frac{\theta_{a^1}\theta_{b^1|a^1}}{\theta_{a^0}\theta_{b^0|a^1}}.$$

Thus, if this representation was faithful for the intended interpretation of the parameter values, we would have $Z(\boldsymbol{\theta}) = \frac{1}{\theta_{a^0}\theta_{b^0|a^1}}$. On the other hand,

$$P_{\boldsymbol{\theta}}(a^0, b^0) = \frac{1}{Z(\boldsymbol{\theta})},$$

which requires that $Z(\boldsymbol{\theta}) = \frac{1}{\theta_{a^0}\theta_{b^0|a^0}}$ in order to be faithful to the desired distribution. Because these two constants are, in general, not equal, we conclude that this representation cannot be faithful to the original Bayesian network. ■

The failure in this example is that the global normalization constant cannot play the role of a local normalization constant within each conditional distribution. This implies that to have an exponential representation of a Bayesian network, we need to ensure that each CPD is locally

normalized. For every exponential CPD this is easy to do. We simply increase the dimension of τ by adding another dimension that has a constant value, say 1. Then the matching element of $\mathtt{t}(\boldsymbol{\theta})$ can be the logarithm of the partition function. This is essentially what we did in example 8.8.

We still might wonder whether a Bayesian network defines a linear exponential family.

Example 8.10

Consider the network structure $A \rightarrow C \leftarrow B$, with binary variables. Assuming a representation that captures general CPDs, our sufficient statistics need to include features that distinguish between the following four assignments:

$$\begin{aligned}\xi_1 &= \langle a^1, b^1, c^1\rangle \\ \xi_2 &= \langle a^1, b^0, c^1\rangle \\ \xi_3 &= \langle a^0, b^1, c^1\rangle \\ \xi_4 &= \langle a^0, b^0, c^1\rangle\end{aligned}$$

More precisely, we need to be able to modify the CPD $P(C \mid A, B)$ to change the probability of one of these assignments without modifying the probability of the other three. This implies that $\tau(\xi_1), \ldots, \tau(\xi_4)$ must be linearly independent: otherwise, we could not change the probability of one assignment without changing the others. Because our model is a linear function of the sufficient statistics, we can choose any set of orthogonal basis vectors that we want; in particular, we can assume without loss of generality that the first four coordinates of the sufficient statistics are $\tau_i(\xi) = \mathbf{1}\{\xi = \xi_i\}$, and that any additional coordinates of the sufficient statistics are not linearly dependent on these four. Moreover, since the model is over a finite set of events, any choice of parameters can be normalized. Thus, the space of natural parameters is $\mathcal{R}^K$, where K is dimension of the sufficient statistics vector. The linear family over such features is essentially a Markov network over the clique $\{A, B, C\}$. Thus, the parameterization of this family includes cases where A and B are not independent, violating the independence properties of the Bayesian network. ■

Thus, this simple Bayesian network cannot be represented by a linear family. **More broadly, although a Bayesian network with suitable CPDs defines an exponential family, this family is not generally a linear one. In particular, any network that contains immoralities does not induce a linear exponential family.**

8.4 Entropy and Relative Entropy

We now explore some of the consequences of representation of models in factored form and of their exponential family representation. These both suggest some implications of these representations and will be useful in developments in subsequent chapters.

8.4.1 Entropy

We start with the notion of entropy. Recall that the entropy of a distribution is a measure of the amount of "stochasticity" or "noise" in the distribution. A low entropy implies that most of the distribution mass is on a few instances, while a larger entropy suggests a more uniform distribution. Another interpretation we discussed in appendix A.1 is the number of bits needed, on average, to encode instances in the distribution.

In various tasks we need to compute the entropy of given distributions. As we will see, we also encounter situations where we want to choose a distribution that maximizes the entropy subject to some constraints. A characterization of entropy will allow us to perform both tasks more efficiently.

8.4.1.1 Entropy of an Exponential Model

We now consider the task of computing the entropy for distributions in an an exponential family defined by τ and $\mathtt{t}$.

Theorem 8.1 *Let $P_{\boldsymbol{\theta}}$ be a distribution in an exponential family defined by the functions τ and $\mathtt{t}$. Then*

$$\boldsymbol{H}_{P_{\boldsymbol{\theta}}}(\mathcal{X}) = \ln Z(\boldsymbol{\theta}) - \langle \boldsymbol{E}_{P_{\boldsymbol{\theta}}}[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}) \rangle. \tag{8.7}$$

While this formulation seems fairly abstract, it does provide some insight. The entropy decomposes as a difference of two terms. The first is the partition function $Z(\boldsymbol{\theta})$. The second depends only on the *expected value* of the sufficient statistics $\tau(\mathcal{X})$. Thus, instead of considering each assignment to $\mathcal{X}$, we need to know only the expectations of the statistics under $P_{\boldsymbol{\theta}}$. As we will see, this is a recurring theme in our discussion of exponential families.

Example 8.11 *We now apply this result to a Gaussian distribution $X \sim N(\mu, \sigma^2)$, as formulated in the exponential family in example 8.3. Plugging into equation (8.7) the definitions of τ, $\mathtt{t}$, and Z from equation (8.4), equation (8.5), and equation (8.6), respectively, we get*

$$\begin{aligned}
\boldsymbol{H}_P(X) &= \frac{1}{2}\ln 2\pi\sigma^2 + \frac{\mu^2}{2\sigma^2} - \frac{2\mu}{2\sigma^2}\boldsymbol{E}_P[X] + \frac{1}{2\sigma^2}\boldsymbol{E}_P\left[X^2\right] \\
&= \frac{1}{2}\ln 2\pi\sigma^2 + \frac{\mu^2}{2\sigma^2} - \frac{2\mu}{2\sigma^2}\mu + \frac{1}{2\sigma^2}(\sigma^2 + \mu^2) \\
&= \frac{1}{2}\ln 2\pi e\sigma^2
\end{aligned}$$

where we used the fact that $\boldsymbol{E}_P[X] = \mu$ and $\boldsymbol{E}_P\left[X^2\right] = \mu^2 + \sigma^2$. ■

We can apply the formulation of theorem 8.1 directly to write the entropy of a Markov network.

Proposition 8.1 *If $P(\mathcal{X}) = \frac{1}{Z}\prod_k \phi_k(\boldsymbol{D}_k)$ is a Markov network, then*

$$\boldsymbol{H}_P(\mathcal{X}) = \ln Z + \sum_k \boldsymbol{E}_P[-\ln \phi_k(\boldsymbol{D}_k)].$$

Example 8.12 *Consider a simple Markov network with two potentials $\beta_1(A, B)$ and $\beta_2(B, C)$, so that*

		$\beta_1(A,B)$
a^0	b^0	2
a^0	b^1	1
a^1	b^0	1
a^1	b^1	5

		$\beta_2(B,C)$
b^0	c^0	6
b^0	c^1	1
b^1	c^0	1
b^1	c^1	0.5

Simple calculations show that $Z = 30$, and the marginal distributions are

A	B	$P(A,B)$
a^0	b^0	0.47
a^0	b^1	0.05
a^1	b^0	0.23
a^1	b^1	0.25

B	C	$P(B,C)$
b^0	c^0	0.6
b^0	c^1	0.1
b^1	c^0	0.2
b^1	c^1	0.1

Using proposition 8.1, we can calculate the entropy:

$$\begin{aligned}
\boldsymbol{H}_P(A,B,C) &= \ln Z + \boldsymbol{E}_P[-\ln \beta_1(A,B)] + \boldsymbol{E}_P[-\ln \beta_2(B,C)] \\
&= \ln Z \\
&\quad -P(a^0,b^0)\ln \beta_1(a^0,b^0) - P(a^0,b^1)\ln \beta_1(a^0,b^1) \\
&\quad -P(a^1,b^0)\ln \beta_1(a^1,b^0) - P(a^1,b^1)\ln \beta_1(a^1,b^1) \\
&\quad -P(b^0,c^0)\ln \beta_2(b^0,c^0) - P(b^0,c^1)\ln \beta_2(b^0,c^1) \\
&\quad -P(b^1,c^0)\ln \beta_2(b^1,c^0) - P(b^1,c^1)\ln \beta_2(b^1,c^1) \\
&= 3.4012 \\
&\quad -0.47 * 0.69 - 0.05 * 0 - 0.23 * 0 - 0.25 * 1.60 \\
&\quad -0.6 * 1.79 - 0.1 * 0 - 0.2 * 0 - 0.1 * -0.69 \\
&= 1.670.
\end{aligned}$$

■

In this example, the number of terms we evaluated is the same as what we would have considered using the original formulation of the entropy where we sum over all possible joint assignments. However, if we consider more complex networks, the number of joint assignments is exponentially large while the number of potentials is typically reasonable, and each one involves the joint assignments to only a few variables.

Note, however, that to use the formulation of proposition 8.1 we need to perform a global computation to find the value of the partition function Z as well as the marginal distribution over the scope of each potential $\boldsymbol{D}_k$. As we will see in later chapters, in some network structures, these computations can be done efficiently.

Terms such as $\boldsymbol{E}_P[-\ln \beta_k(\boldsymbol{D}_k)]$ resemble the entropy of $\boldsymbol{D}_k$. However, since the marginal over $\boldsymbol{D}_k$ is usually not identical to the potential β_k, such terms are not entropy terms. In some sense we can think of $\ln Z$ as a correction for this discrepancy. For example, if we multiply all the entries of β_k by a constant c, the corresponding term $\boldsymbol{E}_P[-\ln \beta_k(\boldsymbol{D}_k)]$ will decrease by $\ln c$. However, at the same time $\ln Z$ will increase by the same constant, since it is canceled out in the normalization.

8.4.1.2 Entropy of Bayesian Networks

We now consider the entropy of a Bayesian network. Although we can address this computation using our general result in theorem 8.1, it turns out that the formulation for Bayesian networks is simpler. Intuitively, as we saw, we can represent Bayesian networks as an exponential family where the partition function is 1. This removes the global term from the entropy.

Theorem 8.2 *If $P(\mathcal{X}) = \prod_i P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})$ is a distribution consistent with a Bayesian network $\mathcal{G}$, then*

$$\boldsymbol{H}_P(\mathcal{X}) = \sum_i \boldsymbol{H}_P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})$$

PROOF

$$\begin{aligned}
\boldsymbol{H}_P(\mathcal{X}) &= \boldsymbol{E}_P[-\ln P(\mathcal{X})] \\
&= \boldsymbol{E}_P\left[-\sum_i \ln P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})\right] \\
&= \sum_i \boldsymbol{E}_P\left[-\ln P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})\right] \\
&= \sum_i \boldsymbol{H}_P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}}),
\end{aligned}$$

where the first and last steps invoke the definitions of entropy and conditional entropy. ∎

We see that the entropy of a Bayesian network decomposes as a sum of conditional entropies of the individual conditional distributions. This representation suggests that the entropy of a Bayesian network can be directly "read off" from the CPDs. This impression is misleading. Recall that the conditional entropy term $\boldsymbol{H}_P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})$ can be written as a weighted average of simpler entropies of conditional distributions

$$\boldsymbol{H}_P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}}) = \sum_{\mathrm{pa}_i^{\mathcal{G}}} P(\mathrm{pa}_i^{\mathcal{G}}) \boldsymbol{H}_P(X_i \mid \mathrm{pa}_i^{\mathcal{G}}).$$

While each of the simpler entropy terms in the summation can be computed based on the CPD entries alone, the weighting term $P(\mathrm{pa}_i^{\mathcal{G}})$ is a marginal over $\mathrm{pa}_i^{\mathcal{G}}$ of the joint distribution, and depends on other CPDs upstream of X_i. Thus, computing the entropy of the network requires that we answer probability queries over the network.

However, based on local considerations alone, we can analyze the amount of entropy introduced by each CPD, and thereby provide bounds on the overall entropy:

Proposition 8.2 *If $P(\mathcal{X}) = \prod_i P(X_i \mid \mathrm{Pa}_i^{\mathcal{G}})$ is a distribution consistent with a Bayesian network $\mathcal{G}$, then*

$$\sum_i \min_{\mathrm{pa}_i^{\mathcal{G}}} \boldsymbol{H}_P(X_i \mid \mathrm{pa}_i^{\mathcal{G}}) \leq \boldsymbol{H}_P(\mathcal{X}) \leq \sum_i \max_{\mathrm{pa}_i^{\mathcal{G}}} \boldsymbol{H}_P(X_i \mid \mathrm{pa}_i^{\mathcal{G}}).$$

Thus, if all the CPDs in a Bayesian network are almost deterministic (low conditional entropy given each parent configuration), then the overall entropy of the network is small. Conversely, if all the CPDs are highly stochastic (high conditional entropy) then the overall entropy of the network is high.

8.4.2 Relative Entropy

A related notion is the relative entropy between models. This measure of distance plays an important role in many of the developments of later chapters.

If we consider the relative entropy between an arbitrary distribution Q and a distribution P_θ within an exponential family, we see that the form of P_θ can be exploited to simplify the form of the relative entropy.

Theorem 8.3 *Consider a distribution Q and a distribution $P_{\boldsymbol{\theta}}$ in an exponential family defined by τ and $\mathtt{t}$. Then*

$$\boldsymbol{D}(Q \| P_{\boldsymbol{\theta}}) = -\boldsymbol{H}_Q(\mathcal{X}) - \langle \boldsymbol{E}_Q[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}) \rangle + \ln Z(\boldsymbol{\theta}).$$

The proof is left as an exercise (exercise 8.2).

We see that the quantities of interest are again the expected sufficient statistics and the partition function. Unlike the entropy, in this case we compute the expectation of the sufficient statistics according to Q.

If both distributions are in the same exponential family, then we can further simplify the form of the relative entropy.

Theorem 8.4 *Consider two distribution $P_{\boldsymbol{\theta}_1}$ and $P_{\boldsymbol{\theta}_2}$ within the same exponential family. Then*

$$\boldsymbol{D}(P_{\boldsymbol{\theta}_1} \| P_{\boldsymbol{\theta}_2}) = \langle \boldsymbol{E}_{P_{\boldsymbol{\theta}_1}}[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}_1) - \mathtt{t}(\boldsymbol{\theta}_2) \rangle - \ln \frac{Z(\boldsymbol{\theta}_1)}{Z(\boldsymbol{\theta}_2)}$$

Proof Combine theorem 8.3 with theorem 8.1. ■

When we consider Bayesian networks, we can use the fact that the partition function is constant to simplify the terms in both results.

Theorem 8.5 *If P is a distribution consistent with a Bayesian network $\mathcal{G}$, then*

$$\boldsymbol{D}(Q \| P) = -\boldsymbol{H}_Q(\mathcal{X}) - \sum_i \sum_{\mathrm{pa}_i^{\mathcal{G}}} Q(\mathrm{pa}_i^{\mathcal{G}}) \boldsymbol{E}_{Q(X_i \mid \mathrm{pa}_i^{\mathcal{G}})} \left[\ln P(X_i \mid \mathrm{pa}_i^{\mathcal{G}}) \right];$$

If Q is also consistent with $\mathcal{G}$, then

$$\boldsymbol{D}(Q \| P) = \sum_i \sum_{\mathrm{pa}_i^{\mathcal{G}}} Q(\mathrm{pa}_i^{\mathcal{G}}) \boldsymbol{D}(Q(X_i \mid \mathrm{pa}_i^{\mathcal{G}}) \| P(X_i \mid \mathrm{pa}_i^{\mathcal{G}})).$$

The second result shows that, analogously to the form of the entropy of Bayesian networks, we can write the relative entropy between two distributions consistent with $\mathcal{G}$ as a weighted sum of the relative entropies between the conditional distributions. These conditional relative entropies can be evaluated directly using the CPDs of the two networks. The weighting of these relative entropies depends on the the joint distribution Q.

8.5 Projections

projection

As we discuss in appendix A.1.3, we can view the relative entropy as a notion of distance between two distributions. We can therefore use it as the basis for an important operation — the *projection* operation — which we will utilize extensively in subsequent chapters. Similar to the geometric concept of projecting a point onto a hyperplane, we consider the problem of finding the distribution, within a given exponential family, that is closest to a given distribution

in terms of relative entropy. For example, we want to perform such a projection when we approximate a complex distribution with one with a simple structure. As we will see, this is a crucial strategy for approximate inference in networks where exact inference is infeasible. In such an approximation we would like to find the best (that is, closest) approximation within a family in which we can perform inference. Moreover, the problem of *learning* a graphical model can also be posed as a projection problem of the empirical distribution observed in the data onto a desired family.

Suppose we have a distribution P and we want to approximate it with another distribution Q in a class of distributions $\mathcal{Q}$ (for example, an exponential family). For example, we might want to approximate P with a product of marginal distributions. Because the notion of relative entropy is not symmetric, we can use it to define two types of approximations.

Definition 8.4 *Let P be a distribution and let $\mathcal{Q}$ be a convex set of distributions.*

I-projection

- *The* I-projection *(information projection) of P onto $\mathcal{Q}$ is the distribution*

$$Q^I = \arg\min_{Q \in \mathcal{Q}} \boldsymbol{D}(Q\|P).$$

M-projection

- *The* M-projection *(moment projection) of P onto $\mathcal{Q}$ is the distribution*

$$Q^M = \arg\min_{Q \in \mathcal{Q}} \boldsymbol{D}(P\|Q).$$

■

8.5.1 Comparison

We can think of both Q^I and Q^M as the projection of P into the set $\mathcal{Q}$ in the sense that it is the distribution closest to P. Moreover, if $P \in \mathcal{Q}$, then in both definitions the projection would be P. However, because the relative entropy is not symmetric, these two projections are, in general, different. To understand the differences between these two projections, let us consider a few examples.

Example 8.13 *Suppose we have a non-Gaussian distribution P over the reals. We can consider the M-projection and the I-projection on the family of Gaussian distributions. As a concrete example, consider the distribution P of figure 8.1. As we can see, the two projections are different Gaussian distributions. (The M-projection was found using the analytic form that we will discuss, and the I-projection by gradient ascent in the (μ, σ^2) space.) Although the means of the two projected distributions are relatively close, the M-projection has larger variance than the I-projection.* ■

We can better understand these differences if we examine the objective function optimized by each projection. Recall that the M-projection Q^M minimizes

$$\boldsymbol{D}(P\|Q) = -\boldsymbol{H}_P(X) + \boldsymbol{E}_P[-\ln Q(X)].$$

We see that, in general, we want Q^M to have high density in regions that are probable according to P, since a small $-\ln Q(X)$ in these regions will lead to a smaller second term. At the same time, there is a high penalty for assigning low density to regions where $P(X)$ is nonnegligible.

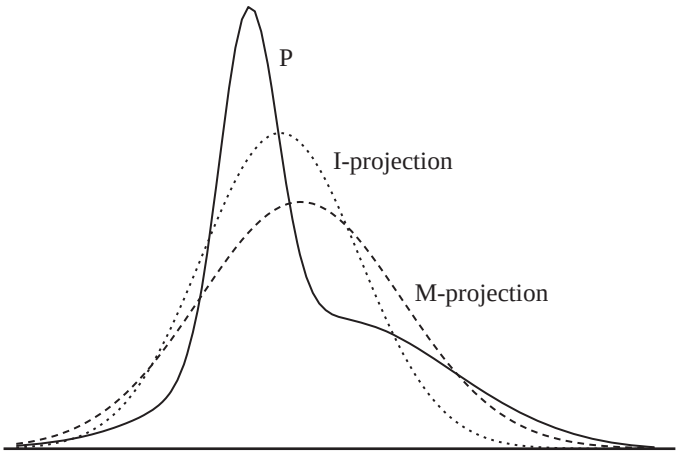


Figure 8.1 Example of M- and I-projections into the family of Gaussian distributions

As a consequence, although the M-projection attempts to match the main mass of P, its high variance is a compromise to ensure that it assigns reasonably high density to all regions that are in the support of P.

On the other hand, the I-projection minimizes

$$\boldsymbol{D}(Q\|P) = -\boldsymbol{H}_Q(X) + \boldsymbol{E}_Q[-\ln P(X)].$$

Thus, the first term incurs a penalty for low entropy, which in the case of a Gaussian Q translates to a penalty on small variance. The second term, $\boldsymbol{E}_Q[-\ln P(X)]$, encodes a preference for assigning higher density to regions where $P(X)$ is large and very low density to regions where $P(X)$ is small. Without the first term, we can minimize the second by putting all of the mass of Q on the most probable point according to P. The compromise between the two terms results in the distribution we see in figure 8.1.

A similar phenomenon occurs in discrete distributions.

Example 8.14

Now consider the projection of a distribution $P(A, B)$ onto the family of factored distributions $Q(A, B) = Q(A)Q(B)$. Suppose $P(A, B)$ is the following distribution:

$$\begin{aligned} P(a^0, b^0) &= 0.45 \\ P(a^0, b^1) &= 0.05 \\ P(a^1, b^0) &= 0.05 \\ P(a^1, b^1) &= 0.45. \end{aligned}$$

That is, the distribution P puts almost all of the mass on the event $A = B$. This distribution is a particularly difficult one to approximate using a factored distribution, since in P the two variables A and B are highly correlated, a dependency that cannot be captured using a fully factored Q.

Again, it is instructive to compare the M-projection and the I-projection of this distribution (see figure 8.2). It follows from example A.7 (appendix A.5.3) that the M-projection of this distribution is

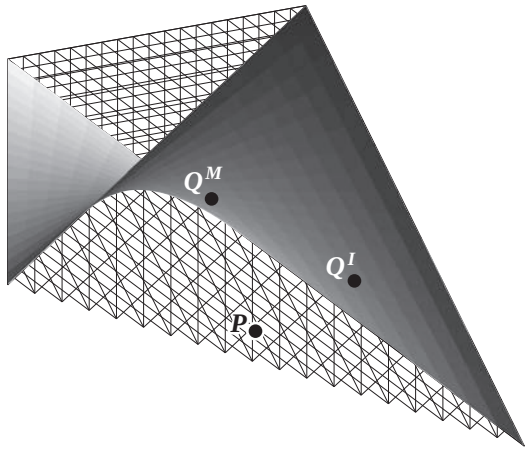


Figure 8.2 Example of M- and I-projections of a two variable discrete distribution where $P(a^0 = b^0) = P(a^1 = b^1) = 0.45$ and $P(a^0 = b^1) = P(a^0 = b^1) = 0.05$ onto factorized distribution. Each axis denotes the probability of an instance: $P(a^1, b^1)$, $P(a^1, b^0)$, and $P(a^0, b^1)$. The wire surfaces mark the region of legal distributions. The solid surface shows the distributions where A and independent of B. The points show P and its two projections.

the uniform distribution:

$$\begin{aligned} Q^M(a^0, b^0) &= 0.5 * 0.5 = 0.25 \\ Q^M(a^0, b^1) &= 0.5 * 0.5 = 0.25 \\ Q^M(a^1, b^0) &= 0.5 * 0.5 = 0.25 \\ Q^M(a^1, b^1) &= 0.5 * 0.5 = 0.25. \end{aligned}$$

In contrast, the I-projection focuses on one of the two "modes" of the distribution, either when both A and B are true or when both are false. Since the distribution is symmetric about these modes, there are two I-projections. One of them is

$$\begin{aligned} Q^I(a^0, b^0) &= 0.25 * 0.25 = 0.0625 \\ Q^I(a^0, b^1) &= 0.25 * 0.75 = 0.1875 \\ Q^I(a^1, b^0) &= 0.75 * 0.25 = 0.1875 \\ Q^I(a^1, b^1) &= 0.75 * 0.75 = 0.5625. \end{aligned}$$

The second I-projection is symmetric around the opposite mode a^0, b^0. ■

As in example 8.13, we can understand these differences by considering the underlying mathematics. **The M-projection attempts to give all assignments reasonably high probability, whereas the I-projection attempts to focus on high-probability assignments in P while maintaining a reasonable entropy.** In this case, this behavior results in a uniform distribution for the M-projection, whereas the I-projection places most of the probability mass on one of the two assignments where P has high probability.

8.5.2 M-Projections

Can we say more about the form of these projections? We start by considering M-projections onto a simple family of distributions.

Proposition 8.3 *Let P be a distribution over $X_1, \ldots, X_n$, and let $\mathcal{Q}$ be the family of distributions consistent with $\mathcal{G}_\emptyset$, the empty graph. Then*

$$Q^M = \arg\min_{Q \models \mathcal{G}_\emptyset} \boldsymbol{D}(P\|Q)$$

is the distribution:

$$Q^M(X_1, \ldots, X_n) = P(X_1)P(X_2)\cdots P(X_n).$$

Proof Consider a distribution $Q \models \mathcal{G}_\emptyset$. Since Q factorizes, we can rewrite $\boldsymbol{D}(P\|Q)$:

$$\begin{aligned}
\boldsymbol{D}(P\|Q) &= \boldsymbol{E}_P[\ln P(X_1, \ldots, X_n) - \ln Q(X_1, \ldots, X_n)] \\
&= \boldsymbol{E}_P[\ln P(X_1, \ldots, X_n)] - \sum_i \boldsymbol{E}_P[\ln Q(X_i)] \\
&= \boldsymbol{E}_P\left[\ln \frac{P(X_1, \ldots, X_n)}{P(X_1)\cdots P(X_n)}\right] + \sum_i \boldsymbol{E}_P\left[\ln \frac{P(X_i)}{Q(X_i)}\right] \\
&= \boldsymbol{D}(P\|Q^M) + \sum_i \boldsymbol{D}(P(X_i)\|Q(X_i)) \\
&\geq \boldsymbol{D}(P\|Q^M).
\end{aligned}$$

The last step relies on the nonnegativity of the relative entropy. We conclude that $\boldsymbol{D}(P\|Q) \geq \boldsymbol{D}(P\|Q^M)$ with equality only if $Q(X_i) = P(X_i)$ for all i. That is, only when $Q = Q^M$. ■

Hence, the M-projection of P onto factored distribution is simply the product of marginals of P.

This theorem is an instance of a much more general result. To understand the generalization, we observe that the family $\mathcal{Q}$ of fully factored distributions is characterized by a vector of sufficient statistics that simply counts, for each variable X_i, the number of occurrences of each of its values. The marginal distributions over the X_i's are simply the expectations, relative to P, of these sufficient statistics. We see that, by selecting Q to match these expectations, we obtain the M-projection.

As we now show, this is not an accident. The characterization of a distribution P that is relevant to computing its M-projection into $\mathcal{Q}$ is precisely the expectation, relative to P, of the sufficient statistic function of $\mathcal{Q}$.

Theorem 8.6

Let P be a distribution over $\mathcal{X}$, and let $\mathcal{Q}$ be an exponential family defined by the functions $\tau(\xi)$ and $\mathtt{t}(\boldsymbol{\theta})$. If there is a set of parameters $\boldsymbol{\theta}$ such that $\boldsymbol{E}_{Q_{\boldsymbol{\theta}}}[\tau(\mathcal{X})] = \boldsymbol{E}_P[\tau(\mathcal{X})]$, then the M-projection of P is $Q_{\boldsymbol{\theta}}$.

PROOF Suppose that $\boldsymbol{E}_P[\tau(\mathcal{X})] = \boldsymbol{E}_{Q_{\boldsymbol{\theta}}}[\tau(\mathcal{X})]$, and let $\boldsymbol{\theta}'$ be some set of parameters. Then,

$$\begin{aligned}
\boldsymbol{D}(P\|Q_{\boldsymbol{\theta}'}) - \boldsymbol{D}(P\|Q_{\boldsymbol{\theta}}) &= -\boldsymbol{H}_P(\mathcal{X}) - \langle \boldsymbol{E}_P[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}')\rangle + \ln Z(\boldsymbol{\theta}') \\
&\quad + \boldsymbol{H}_P(\mathcal{X}) + \langle \boldsymbol{E}_P[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta})\rangle - \ln Z(\boldsymbol{\theta}) \\
&= \langle \boldsymbol{E}_P[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}) - \mathtt{t}(\boldsymbol{\theta}')\rangle - \ln \frac{Z(\boldsymbol{\theta})}{Z(\boldsymbol{\theta}')} \\
&= \langle \boldsymbol{E}_{Q_{\boldsymbol{\theta}}}[\tau(\mathcal{X})], \mathtt{t}(\boldsymbol{\theta}) - \mathtt{t}(\boldsymbol{\theta}')\rangle - \ln \frac{Z(\boldsymbol{\theta})}{Z(\boldsymbol{\theta}')} \\
&= \boldsymbol{D}(Q_{\boldsymbol{\theta}}\|Q_{\boldsymbol{\theta}'}) \geq 0.
\end{aligned}$$

We conclude that the M-projection of P is $Q_{\boldsymbol{\theta}}$. ∎

This theorem suggests that we can consider both the distribution P and the distributions in $\mathcal{Q}$ in terms of the expectations of $\tau(\mathcal{X})$. Thus, instead of describing a distribution in the family by the set of parameters, we can describe it in terms of the *expected sufficient statistics*. To formalize this intuition, we need some additional notation. We define a mapping from legal parameters in Θ to vectors of sufficient statistics

expected sufficient statistics

$$ess(\boldsymbol{\theta}) = \boldsymbol{E}_{Q_{\boldsymbol{\theta}}}[\tau(\mathcal{X})].$$

Theorem 8.6 shows that if $\boldsymbol{E}_P[\tau(\mathcal{X})]$ is in the image of *ess*, then the M-projection of P is the distribution $Q_{\boldsymbol{\theta}}$ that matches the expected sufficient statistics of P. In other words,

$$\boldsymbol{E}_{Q^M}[\tau(\mathcal{X})] = \boldsymbol{E}_P[\tau(\mathcal{X})].$$

moment matching

This result explains why M-projection is also referred to as *moment matching*. In many exponential families the sufficient statistics are moments (mean, variance, and so forth) of the distribution. In such cases, the M-projection of P is the distribution in the family that matches these moments in P.

We illustrate these concepts in figure 8.3. As we can see, the mapping $ess(\boldsymbol{\theta})$ directly relates parameters to expected sufficient statistics. By comparing the expected sufficient statistics of P to these of distributions in $\mathcal{Q}$, we can find the M-projection.

Moreover, using theorem 8.6, we obtain a general characterization of the M-projection function M-project(s), which maps a vector of expected sufficient statistics to a parameter vector:

Corollary 8.1

Let $\boldsymbol{s}$ be a vector. If $\boldsymbol{s} \in$ image(ess) and ess is invertible, then

$$\text{M-project}(\boldsymbol{s}) = ess^{-1}(\boldsymbol{s}).$$

That is, the parameters of the M-projection of P are simply the inverse of the *ess* mapping, applied to the expected sufficient statistic vector of P. This result allows us to describe the M-projection operation in terms of a specific function. This result assumes, of course, that $\boldsymbol{E}_P[\tau]$ is in the image of *ess* and that *ess* is invertible. In many examples that we consider, the image of *ess* includes all possible vectors of expected sufficient statistics we might encounter. Moreover, if the parameterization is nonredundant, then *ess* is invertible.

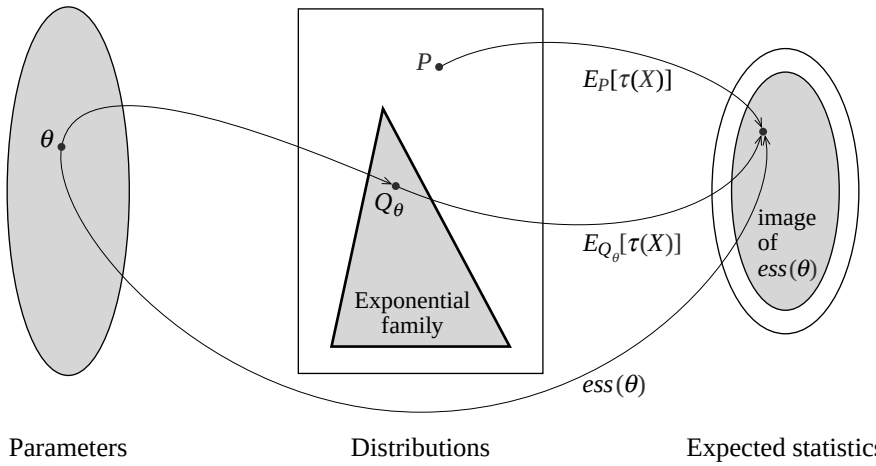


Figure 8.3 Illustration of the relations between parameters, distributions and expected sufficient statistics. Each parameter corresponds to a distribution, which in turn corresponds to a value of the expected statistics. The function *ess* maps parameters directly to expected statistics. If the expected statistics of P and $Q_{\boldsymbol{\theta}}$ match, then $Q_{\boldsymbol{\theta}}$ is the M-projection of P.

Example 8.15

Consider the exponential family of Gaussian distributions. Recall that the sufficient statistics function for this family is $\tau(x) = \langle x, x^2 \rangle$. Given parameters $\boldsymbol{\theta} = \langle \mu, \sigma^2 \rangle$, the expected value of τ is

$$ess(\langle \mu, \sigma^2 \rangle) = \mathbf{E}_{Q_{\langle \mu, \sigma^2 \rangle}}[\tau(X)] = \langle\ \mu, \sigma^2 + \mu^2 \rangle.$$

It is not difficult to show that, for any distribution P, $\mathbf{E}_P[\tau(X)]$ must be in the image of this function (see exercise 8.4). Thus, for any choice of P, we can apply theorem 8.6.

Finally, we can easily invert this function:

$$\text{M-project}(\langle s_1, s_2 \rangle) = ess^{-1}(\langle s_1, s_2 \rangle) = \langle s_1, s_2 - s_1^2 \rangle.$$

Recall that $s_1 = \mathbf{E}_P[X]$ and $s_2 = \mathbf{E}_P[X^2]$. Thus, the estimated parameters are the mean and variance of X according to P, as we would expect. ■

This example shows that the "naive" choice of Gaussian distribution, obtained by matching the mean and variance of a variable X, provides the best Gaussian approximation (in the M-projection sense) to a non-Gaussian distribution over X. We have also provided a solution to the M-projection problem in the case of a factored product of multinomials, in proposition 8.3, which can be viewed as a special case of theorem 8.6. In a more general application of this result, we show in section 11.4.4 a general result on the form of the M-projection for a linear exponential family over discrete state space, including the class of Markov networks.

The analysis for other families of distributions can be subtler.

Example 8.16

We now consider a more complex example of M-projection onto a chain network. Suppose we have a distribution P over variables $X_1, \ldots, X_n$, and want to project it onto the family of distributions Q of the distributions that are consistent with the network structure $X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_n$.

What are the sufficient statistics for this network? Based on our previous discussion, we see that each conditional distribution $Q(X_{i+1} \mid X_i)$ requires a statistic of the form

$$\tau_{x_i, x_{i+1}}(\xi) = \mathbf{I}\{X_i = x_i, X_{i+1} = x_{i+1}\} \ \forall \langle x_i, x_{i+1} \rangle \in Val(X_i) \times Val(X_{i+1}).$$

These statistics are sufficient but are redundant. To see this, note that the "marginal statistics" must agree. That is,

$$\sum_{x_i} \tau_{x_i, x_{i+1}}(\xi) = \sum_{x_{i+2}} \tau_{x_{i+1}, x_{i+2}}(\xi) \ \forall x_{i+1} \in Val(X_{i+1}). \tag{8.8}$$

Although this representation is redundant, we can still apply the mechanisms discussed earlier and consider the function ess that maps parameters of such a network to the sufficient statistics. The expectation of an indicator function is the marginal probability of that event, so that

$$\mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\tau_{x_i, x_{i+1}}(\mathcal{X})\right] = Q_{\boldsymbol{\theta}}(x_i, x_{i+1}).$$

Thus, the function ess simply maps from $\boldsymbol{\theta}$ to the pairwise marginals of consecutive variables in $Q_{\boldsymbol{\theta}}$. Because these are pairwise marginals of an actual distribution, it follows that these sufficient statistics satisfy the consistency constraints of equation (8.8).

How do we invert this function? Given the statistics from P, we want to find a distribution Q that matches them. We start building Q along the structure of the chain. We choose $Q(X_1)$ and $Q(X_2 \mid X_1)$ so that $Q(x_1, x_2) = \mathbf{E}_P[\tau_{x_1, x_2}(\mathcal{X})] = P(x_1, x_2)$. In fact, there is a unique choice that satisfies this equality, where $Q(X_1, X_2) = P(X_1, X_2)$. This choice implies that the marginal distribution $Q(X_2)$ matches the marginal distribution $P(X_2)$. Now, consider our choice of $Q(X_3 \mid X_2)$. We need to ensure that

$$Q(x_3, x_2) = \mathbf{E}_P[\tau_{x_2, x_3}(\mathcal{X})] = P(x_2, x_3).$$

We note that, because $Q(x_3, x_2) = Q(x_3 \mid x_2)Q(x_2) = Q(x_3 \mid x_2)P(x_2)$, we can achieve this equality by setting $Q(x_3 \mid x_2) = P(x_3 \mid x_2)$. Moreover, this implies that $Q(x_3) = P(x_3)$. We can continue this construction recursively to set

$$Q(x_{i+1} \mid x_i) = P(x_{i+1} \mid x_i).$$

Using the preceding argument, we can show that this choice will match the sufficient statistics of P. This suffices to show that this Q is the M-projection of P.

Note that, although this choice of Q coincides with P on pairwise marginals of consecutive variables, it does not necessarily agree with P on other marginals. As an extreme example, consider a distribution P where X_1 and X_3 are identical and both are independent of X_2. If we project this distribution onto a distribution Q with the structure $X_1 \rightarrow X_2 \rightarrow X_3$, then P and Q will not necessarily agree on the joint marginals of X_1, X_3. In Q this distribution will be

$$Q(x_1, x_3) = \sum_{x_2} Q(x_1, x_2)Q(x_3 \mid x_2).$$

Since $Q(x_1, x_2) = P(x_1, x_2) = P(x_1)P(x_2)$ and $Q(x_3 \mid x_2) = P(x_3 \mid x_2) = P(x_3)$, we conclude that $Q(x_1, x_3) = P(x_1)P(x_3)$, losing the equality between X_1 and X_3 in P. ■

This analysis used a redundant parameterization; exercise 8.6 shows how we can reparameterize a directed chain within the linear exponential family and thereby obtain an alternative perspective on the M-projection operation.

So far, all of our examples have had the characteristic that the vector of expected sufficient statistics for a distribution P is always in the image of *ess*; thus, our task has only been to invert *ess*. Unfortunately, there are examples where not every vector of expected sufficient statistics can also be derived from a distribution in our exponential family.

Example 8.17

Consider again the family $\mathcal{Q}$ from example 8.10, of distributions parameterized using network structure $A \rightarrow C \leftarrow B$, with binary variables A, B, C. We can show that the sufficient statistics for this distribution are indicators for all the joint assignments to A, B, and C except one. That is,

$$\begin{aligned}\tau(A,B,C) \quad = \langle \quad & \mathbf{1}\{A = a^1, B = b^1, C = c^1\}, \\ & \mathbf{1}\{A = a^0, B = b^1, C = c^1\}, \\ & \mathbf{1}\{A = a^1, B = b^0, C = c^1\}, \\ & \mathbf{1}\{A = a^1, B = b^1, C = c^0\}, \\ & \mathbf{1}\{A = a^1, B = b^0, C = c^0\}, \\ & \mathbf{1}\{A = a^0, B = b^1, C = c^0\}, \\ & \mathbf{1}\{A = a^0, B = b^0, C = c^1\}\rangle.\end{aligned}$$

If we look at the expected value of these statistics given some member of the family, we have that, since A and B are independent in $Q_{\boldsymbol{\theta}}$, $Q_{\boldsymbol{\theta}}(a^1, b^1) = Q_{\boldsymbol{\theta}}(a^1)Q_{\boldsymbol{\theta}}(b^1)$. Thus, the expected statistics should satisfy

$$\begin{aligned}&\mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^1\}\right] + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^0\}\right] = \\ &\quad \left(\mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^1\}\right] + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^0\}\right]\right. \\ &\qquad \left. + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^0, C = c^1\}\right] + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^0, C = c^0\}\right]\right) \\ &\quad \left(\mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^1\}\right] + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^1, B = b^1, C = c^0\}\right]\right. \\ &\qquad \left. + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^0, B = b^1, C = c^1\}\right] + \mathbf{E}_{Q_{\boldsymbol{\theta}}}\left[\mathbf{1}\{A = a^0, B = b^1, C = c^0\}\right]\right).\end{aligned}$$

This constraint is not typically satisfied by the expected statistics from a general distribution P we might consider projecting. Thus, in this case, there are expected statistics vectors that do not fall within the image of ess. ■

In such cases, and in Bayesian networks in general, the projection procedure is more complex than inverting the *ess* function. Nevertheless, we can show that the projection operation still has an analytic solution.

Theorem 8.7

Let P be a distribution over $X_1, \ldots, X_n$, and let $\mathcal{G}$ be a Bayesian network structure. Then the M-projection Q^M is:

$$Q^M(X_1, \ldots, X_n) = \prod_i P(X_i \mid \mathrm{Pa}_{X_i}^{\mathcal{G}}).$$

Because the mapping *ess* for Bayesian networks is not invertible, the proof of this result (see exercise 8.5) does not build on theorem 8.6 but rather directly on theorem 8.5. This result turns out to be central to our derivation of Bayesian network learning in chapter 17.

8.5.3 I-Projections

What about I-projections? Recall that

$$\boldsymbol{D}(Q\|P) = -\boldsymbol{H}_Q(\mathcal{X}) - \boldsymbol{E}_Q[\ln P(\mathcal{X})].$$

If Q is in some exponential family, we can use the derivation of theorem 8.1 to simplify the entropy term. However, the exponential form of Q does not provide insights into the second term. When dealing with the I-projection of a general distribution P, we are left without further simplifications. However, if the distribution P has some structure, we might be able to simplify $\boldsymbol{E}_Q[\ln P(\mathcal{X})]$ into simpler terms, although the projection problem is still a nontrivial one. We discuss this problem in much more detail in chapter 11.

8.6 Summary

In this chapter, we presented some of the basic technical concepts that underlie many of the techniques we explore in depth later in the book. We defined the formalism of exponential families, which provides the fundamental basis for considering families of related distributions. We also defined the subclass of linear exponential families, which are significantly simpler and yet cover a large fraction of the distributions that arise in practice.

We discussed how the types of distributions described so far in this book fit into this framework, showing that Gaussians, linear Gaussians, and multinomials are all in the linear exponential family. Any class of distributions representable by parameterizing a Markov network of some fixed structure is also in the linear exponential family. By contrast, the class of distributions representable by a Bayesian network of some fixed structure is in the exponential family, but is not in the linear exponential family when the network structure includes an immorality.

We showed how we can use the formulation of an exponential family to facilitate computations such as the entropy of a distribution or the relative entropy between two distributions. The latter computation formed the basis for analyzing a basic operation on distributions: that of projecting a general distribution P into some exponential family $\mathcal{Q}$, that is, finding the distribution within $\mathcal{Q}$ that is closest to P. Because the notion of relative entropy is not symmetric, this concept gave rise to two different definitions: I-projection, where we minimize $\boldsymbol{D}(Q\|P)$, and M-projection, where we minimize $\boldsymbol{D}(P\|Q)$. We analyzed the differences between these two definitions and showed that solving the M-projection problem can be viewed in a particularly elegant way, constructing a distribution Q that matches the expected sufficient statistics (or moments) of P.

As we discuss later in the book, both the I-projection and M-projection turn out to play an important role in graphical models. The M-projection is the formal foundation for addressing the learning problem: there, our goal is to find a distribution in a particular class (for example, a Bayesian network or Markov network of a given structure) that is closest (in the M-projection sense) to the *empirical distribution* observed in a data set from which we wish to learn (see equation (16.4)). The I-projection operation is used when we wish to take a given graphical model P and answer probability queries; when P is too complex to allow queries to be answered

efficiently, one strategy is to construct a simpler distribution Q, which is a good approximation to P (in the I-projection sense).

8.7 Relevant Literature

The concept of exponential families plays a central role in formal statistic theory. Much of the theory is covered by classic textbooks such as Barndorff-Nielsen (1978). See also Lauritzen (1996). Geiger and Meek (1998) discuss the representation of graphical models as exponential families and show that a Bayesian network usually does not define a linear exponential family.

The notion of I-projections was introduced by Csiszàr (1975), who developed the "information geometry" of such projections and their connection to different estimation procedures. In his terminology, M-projections are called "reverse I-projections." The notion of M-projection is closely related to parameter learning, which we revisit in chapter 17 and chapter 20.

8.8 Exercises

Exercise 8.1⋆

Poisson distribution

A variable X with $Val(X) = 0, 1, 2, \ldots$ is *Poisson-distributed* with parameter $\theta > 0$ if

$$P(X = k) = \frac{1}{k!} \exp -\theta\theta^k.$$

This distribution has the property that $\boldsymbol{E}_P[X] = \theta$.

a. Show how to represent the Poisson distribution as a linear exponential family. (Note that unlike most of our running examples, you need to use the auxiliary measure A in the definition.)
b. Use results developed in this chapter to find the entropy of a Poisson distribution and the relative entropy between two Poisson distributions.
c. What is the function *ess* associated with this family? Is it invertible?

Exercise 8.2

Prove theorem 8.3.

Exercise 8.3

In this exercise, we will provide a characterization of when two distributions P_1 and P_2 will have the same M-projection.

a. Let P_1 and P_2 be two distribution over $\mathcal{X}$, and let $\mathcal{Q}$ be an exponential family defined by the functions $\tau(\xi)$ and $\mathtt{t}(\boldsymbol{\theta})$. If $\boldsymbol{E}_{P_1}[\tau(\mathcal{X})] = \boldsymbol{E}_{P_2}[\tau(\mathcal{X})]$, then the M-projection of P_1 and P_2 onto $\mathcal{Q}$ is identical.
b. Now, show that if the function $ess(\boldsymbol{\theta})$ is invertible, then we can prove the converse, showing that the M-projection of P_1 and P_2 is identical only if $\boldsymbol{E}_{P_1}[\tau(\mathcal{X})] = \boldsymbol{E}_{P_2}[\tau(\mathcal{X})]$. Conclude that this is the case for linear exponential families.

Exercise 8.4

Consider the function *ess* for Gaussian variables as described in example 8.15.

a. What is the image of *ess*?
b. Consider terms of the form $\boldsymbol{E}_P[\tau(X)]$ for the Gaussian sufficient statistics from that example. Show that for any distribution P, the expected sufficient statistics is in the image of *ess*.

Exercise 8.5⋆

Prove theorem 8.7. (Hint: Use theorem 8.5.)

Exercise 8.6⋆

Let $X_1, \ldots, X_n$ be binary random variables. Suppose we are given a family $\mathcal{Q}$ of chain distributions of the form $Q(X_1, \ldots, X_n) = Q(X_1)Q(X_2 \mid X_1) \cdots Q(X_n \mid X_{n-1})$. We now show how to reformulate this family as a linear exponential family.

a. Show that the following vector of statistics is sufficient and nonredundant for distributions in the family:

$$\tau(X_1, \ldots, X_n) = \begin{pmatrix} \mathbf{1}\{X_1 = x_1^1\}, \\ \ldots \\ \mathbf{1}\{X_n = x_n^1\}, \\ \mathbf{1}\{X_1 = x_1^1, X_2 = x_2^1\}, \\ \ldots \\ \mathbf{1}\{X_{n-1} = x_{n-1}^1, X_n = x_n^1\} \end{pmatrix}.$$

b. Show that you can reconstruct the distributions $Q(X_1)$ and $Q(X_{i+1} \mid X_i)$ from the the expectation $\mathbf{E}_Q[\tau(X_1, \ldots, X_n)]$. This shows that given the expected sufficient statistics you can reconstruct Q.

c. Suppose you know Q. Show how to reparameterize it as a linear exponential model

$$Q(X_1, \ldots, X_n) = \frac{1}{Z} \exp\left\{ \sum_i \theta_i \mathbf{1}\{X_i = x_i^1\} + \sum_i \theta_{i,i+1} \mathbf{1}\{X_i = x_i^1, X_{i+1} = x_{i+1}^1\} \right\}. \quad (8.9)$$

Note that, because the statistics are sufficient, we know that there are some parameters for which we get equality; the question is to determine their values. Specifically, show that if we choose:

$$\theta_i = \ln \frac{Q(x_1^0, \ldots, x_{i-1}^0, x_i^1, x_{i+1}^0, \ldots, x_n^0)}{Q(x_1^0, \ldots, x_n^0)}$$

and

$$\theta_{i,i+1} = \ln \frac{Q(x_1^0, \ldots, x_{i-1}^0, x_i^1, x_{i+1}^1 x_{i+2}^0, \ldots, x_n^0)}{Q(x_1^0, \ldots, x_n^0)} - \theta_i - \theta_{i+1}$$

then we get equality in equation (8.9) for all assignments to $X_1, \ldots, X_n$.

Part II

Inference

9 Exact Inference: Variable Elimination

In this chapter, we discuss the problem of performing inference in graphical models. We show that the structure of the network, both the conditional independence assertions it makes and the associated factorization of the joint distribution, is critical to our ability to perform inference effectively, allowing tractable inference even in complex networks.

conditional probability query

Our focus in this chapter is on the most common query type: the *conditional probability query*, $P(\boldsymbol{Y} \mid \boldsymbol{E} = \boldsymbol{e})$ (see section 2.1.5). We have already seen several examples of conditional probability queries in chapter 3 and chapter 4; as we saw, such queries allow for many useful reasoning patterns, including explanation, prediction, intercausal reasoning, and many more.

By the definition of conditional probability, we know that

$$P(\boldsymbol{Y} \mid \boldsymbol{E} = \boldsymbol{e}) = \frac{P(\boldsymbol{Y}, \boldsymbol{e})}{P(\boldsymbol{e})}. \tag{9.1}$$

Each of the instantiations of the numerator is a probability expression $P(\boldsymbol{y}, \boldsymbol{e})$, which can be computed by summing out all entries in the joint that correspond to assignments consistent with $\boldsymbol{y}, \boldsymbol{e}$. More precisely, let $\boldsymbol{W} = \mathcal{X} - \boldsymbol{Y} - \boldsymbol{E}$ be the random variables that are neither query nor evidence. Then

$$P(\boldsymbol{y}, \boldsymbol{e}) = \sum_{\boldsymbol{w}} P(\boldsymbol{y}, \boldsymbol{e}, \boldsymbol{w}). \tag{9.2}$$

Because $\boldsymbol{Y}, \boldsymbol{E}, \boldsymbol{W}$ are all of the network variables, each term $P(\boldsymbol{y}, \boldsymbol{e}, \boldsymbol{w})$ in the summation is simply an entry in the joint distribution.

The probability $P(\boldsymbol{e})$ can also be computed directly by summing out the joint. However, it can also be computed as

$$P(\boldsymbol{e}) = \sum_{\boldsymbol{y}} P(\boldsymbol{y}, \boldsymbol{e}), \tag{9.3}$$

which allows us to reuse our computation for equation (9.2). If we compute both equation (9.2) and equation (9.3), we can then divide each $P(\boldsymbol{y}, \boldsymbol{e})$ by $P(\boldsymbol{e})$, to get the desired conditional probability $P(\boldsymbol{y} \mid \boldsymbol{e})$. Note that this process corresponds to taking the vector of marginal probabilities $P(\boldsymbol{y}^1, \boldsymbol{e}), \ldots, P(\boldsymbol{y}^k, \boldsymbol{e})$ (where $k = |Val(\boldsymbol{Y})|$) and *renormalizing* the entries to sum to 1.

renormalization

9.1 Analysis of Complexity

In principle, a graphical model can be used to answer all of the query types described earlier. We simply generate the joint distribution and exhaustively sum out the joint (in the case of a conditional probability query), search for the most likely entry (in the case of a MAP query), or both (in the case of a marginal MAP query). However, this approach to the inference problem is not very satisfactory, since it returns us to the exponential blowup of the joint distribution that the graphical model representation was precisely designed to avoid.

Unfortunately, we now show that **exponential blowup of the inference task is (almost certainly) unavoidable in the worst case: The problem of inference in graphical models is $\mathcal{NP}$-hard, and therefore it probably requires exponential time in the worst case (except in the unlikely event that $\mathcal{P} = \mathcal{NP}$). Even worse, approximate inference is also $\mathcal{NP}$-hard. Importantly, however, the story does not end with this negative result. In general, we care not about the worst case, but about the cases that we encounter in practice. As we show in the remainder of this part of the book, many real-world applications can be tackled very effectively using exact or approximate inference algorithms for graphical models.**

In our theoretical analysis, we focus our discussion on Bayesian networks. Because any Bayesian network can be encoded as a Markov network with no increase in its representation size, a hardness proof for inference in Bayesian networks immediately implies hardness of inference in Markov networks.

9.1.1 Analysis of Exact Inference

To address the question of the complexity of BN inference, we need to address the question of how we encode a Bayesian network. Without going into too much detail, we can assume that the encoding specifies the DAG structure and the CPDs. For the following results, we assume the worst-case representation of a CPD as a full table of size $|Val(\{X_i\} \cup \mathrm{Pa}_{X_i})|$.

As we discuss in appendix A.3.4, most analyses of complexity are stated in terms of decision problems. We therefore begin with a formulation of the inference problem as a decision problem, and then discuss the numerical version. One natural decision version of the conditional probability task is the problem *BN-Pr-DP*, defined as follows:

> Given a Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a variable $X \in \mathcal{X}$, and a value $x \in Val(X)$, decide whether $P_{\mathcal{B}}(X = x) > 0$.

Theorem 9.1 *The decision problem BN-Pr-DP is $\mathcal{NP}$-complete.*

Proof It is straightforward to prove that *BN-Pr-DP* is in $\mathcal{NP}$: In the guessing phase, we guess a full assignment ξ to the network variables. In the verification phase, we check whether $X = x$ in ξ, and whether $P(\xi) > 0$. One of these guesses succeeds if and only if $P(X = x) > 0$. Computing $P(\xi)$ for a full assignment of the network variables requires only that we multiply the relevant entries in the factors, as per the chain rule for Bayesian networks, and hence can be done in linear time.

To prove $\mathcal{NP}$-hardness, we need to show that, if we can answer instances in *BN-Pr-DP*, we can use that as a subroutine to answer questions in a class of problems that is known to be $\mathcal{NP}$-hard. We will use a reduction from the *3-SAT* problem defined in definition A.8.

3-SAT

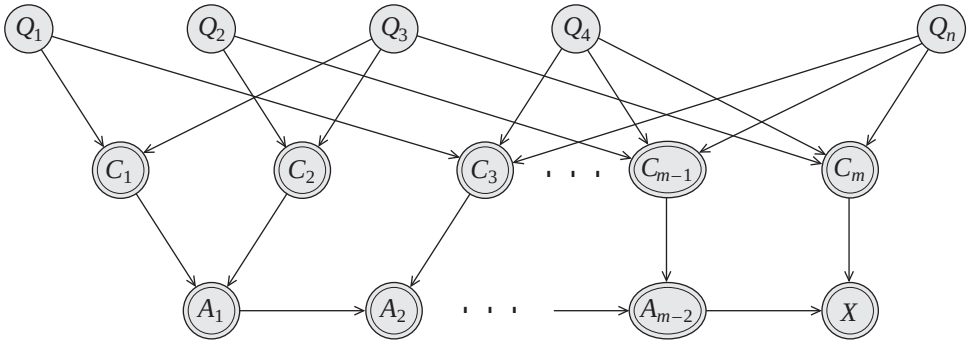


Figure 9.1 An outline of the network structure used in the reduction of 3-SAT to Bayesian network inference.

To show the reduction, we show the following: Given any 3-SAT formula ϕ, we can create a Bayesian network $\mathcal{B}_\phi$ with some distinguished variable X, such that ϕ is satisfiable if and only if $P_{\mathcal{B}_\phi}(X = x^1) > 0$. Thus, if we can solve the Bayesian network inference problem in polynomial time, we can also solve the 3-SAT problem in polynomial time. To enable this conclusion, our BN $\mathcal{B}_\phi$ has to be constructible in time that is polynomial in the length of the formula ϕ.

Consider a 3-SAT instance ϕ over the propositional variables $q_1, \ldots, q_n$. Figure 9.1 illustrates the structure of the network constructed in this reduction. Our Bayesian network $\mathcal{B}_\phi$ has a node Q_k for each propositional variable q_k; these variables are roots, with $P(q_k^1) = 0.5$. It also has a node C_i for each clause C_i. There is an edge from Q_k to C_i if q_k or $\neg q_k$ is one of the literals in C_i. The CPD for C_i is deterministic, and chosen such that it exactly duplicates the behavior of the clause. Note that, because C_i contains at most three variables, the CPD has at most eight distributions, and at most sixteen entries.

We want to introduce a variable X that has the value 1 if and only if all the C_i's have the value 1. We can achieve this requirement by having $C_1, \ldots, C_m$ be parents of X. This construction, however, has the property that $P(X \mid C_1, \ldots, C_m)$ is exponentially large when written as a table. To avoid this difficulty, we introduce intermediate "AND" gates $A_1, \ldots, A_{m-2}$, so that A_1 is the "AND" of C_1 and C_2, A_2 is the "AND" of A_1 and C_3, and so on. The last variable X is the "AND" of A_{m-2} and C_m. This construction achieves the desired effect: X has value 1 if and only if all the clauses are satisfied. Furthermore, in this construction, all variables have at most three (binary-valued) parents, so that the size of $\mathcal{B}_\phi$ is polynomial in the size of ϕ.

It follows that $P_{\mathcal{B}_\phi}(x^1 \mid q_1, \ldots, q_n) = 1$ if and only if $q_1, \ldots, q_n$ is a satisfying assignment for ϕ. Because the prior probability of each possible assignment is $1/2^n$, we get that the overall probability $P_{\mathcal{B}_\phi}(x^1)$ is the number of satisfying assignments to ϕ, divided by 2^n. We can therefore test whether ϕ has a satisfying assignment simply by checking whether $P(x^1) > 0$. ■

This analysis shows that the decision problem associated with Bayesian network inference is $\mathcal{NP}$-complete. However, the problem is originally a numerical problem. Precisely the same construction allows us to provide an analysis for the original problem formulation. We define the problem *BN-Pr* as follows:

Given: a Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a variable $X \in \mathcal{X}$, and a value $x \in Val(X)$,
compute $P_{\mathcal{B}}(X = x)$.

Our task here is to compute the total probability of network instantiations that are consistent with $X = x$. Or, in other words, to do a weighted count of instantiations, with the weight being the probability. An appropriate complexity class for counting problems is $\#\mathcal{P}$: Whereas $\mathcal{NP}$ represents problems of deciding "are there any solutions that satisfy certain requirements," $\#\mathcal{P}$ represents problems that ask "how many solutions are there that satisfy certain requirements." It is not surprising that we can relate the complexity of the BN inference problem to the counting class $\#\mathcal{P}$:

Theorem 9.2 *The problem BN-Pr is $\#\mathcal{P}$-complete.*

We leave the proof as an exercise (exercise 9.1).

9.1.2 Analysis of Approximate Inference

Upon noting the hardness of exact inference, a natural question is whether we can circumvent the difficulties by compromising, to some extent, on the accuracies of our answers. Indeed, in many applications we can tolerate some imprecision in the final probabilities: it is often unlikely that a change in probability from 0.87 to 0.92 will change our course of action. Thus, we now explore the computational complexity of approximate inference.

To analyze the approximate inference task formally, we must first define a metric for evaluating the quality of our approximation. We can consider two perspectives on this issue, depending on how we choose to define our query. Consider first our previous formulation of the conditional probability query task, where our goal is to compute the probability $P(\boldsymbol{Y} \mid \boldsymbol{e})$ for some set of variables $\boldsymbol{Y}$ and evidence $\boldsymbol{e}$. The result of this type of query is a probability distribution over $\boldsymbol{Y}$. Given an approximate answer to this query, we can evaluate its quality using any of the distance metrics we define for probability distributions in appendix A.1.3.3.

There is, however, another way of looking at this task, one that is somewhat simpler and will be very useful for analyzing its complexity. Consider a *specific* query $P(\boldsymbol{y} \mid \boldsymbol{e})$, where we are focusing on one particular assignment $\boldsymbol{y}$. The approximate answer to this query is a number ρ, whose accuracy we wish to evaluate relative to the correct probability. One way of evaluating the accuracy of an estimate is as simple as the difference between the approximate answer and the right one.

Definition 9.1 absolute error

An estimate ρ has absolute error ϵ *for $P(\boldsymbol{y} \mid \boldsymbol{e})$ if:*

$$|P(\boldsymbol{y} \mid \boldsymbol{e}) - \rho| \leq \epsilon.$$

■

This definition, although plausible, is somewhat weak. Consider, for example, a situation in which we are trying to compute the probability of a really rare disease, one whose true probability is, say, 0.00001. In this case, an absolute error of 0.0001 is unacceptable, even though such an error may be an excellent approximation for an event whose probability is 0.3. A stronger definition of accuracy takes into consideration the value of the probability that we are trying to estimate:

Definition 9.2 relative error

An estimate ρ has relative error *ϵ for $P(\boldsymbol{y} \mid \boldsymbol{e})$ if:*

$$\frac{\rho}{1+\epsilon} \leq P(\boldsymbol{y} \mid \boldsymbol{e}) \leq \rho(1+\epsilon).$$

■

Note that, unlike absolute error, relative error makes sense even for $\epsilon > 1$. For example, $\epsilon = 4$ means that $P(\boldsymbol{y} \mid \boldsymbol{e})$ is at least 20 percent of ρ and at most 600 percent of ρ. For probabilities, where low values are often very important, relative error appears much more relevant than absolute error.

With these definitions, we can turn to answering the question of whether approximate inference is actually an easier problem. A priori, it seems as if the extra slack provided by the approximation might help. Unfortunately, this hope turns out to be unfounded. As we now show, approximate inference in Bayesian networks is also $\mathcal{NP}$-hard.

This result is straightforward for the case of relative error.

Theorem 9.3

The following problem is $\mathcal{NP}$-hard:

> *Given a Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a variable $X \in \mathcal{X}$, and a value $x \in Val(X)$, find a number ρ that has relative error ϵ for $P_{\mathcal{B}}(X = x)$.*

Proof The proof is obvious based on the original $\mathcal{NP}$-hardness proof for exact Bayesian network inference (theorem 9.1). There, we proved that it is $\mathcal{NP}$-hard to decide whether $P_{\mathcal{B}}(x^1) > 0$. Now, assume that we have an algorithm that returns an estimate ρ to the same $P_{\mathcal{B}}(x^1)$, which is guaranteed to have relative error ϵ for some $\epsilon > 0$. Then $\rho > 0$ if and only if $P_{\mathcal{B}}(x^1) > 0$. Thus, achieving this relative error is as $\mathcal{NP}$-hard as the original problem. ■

We can generalize this result to make $\epsilon(n)$ a function that grows with the input size n. Thus, for example, we can define $\epsilon(n) = 2^{2^n}$ and the theorem still holds. Thus, in a sense, this result is not so interesting as a statement about hardness of approximation. Rather, it tells us that relative error is too strong a notion of approximation to use in this context.

What about absolute error? As we will see in section 12.1.2, the problem of just approximating $P(X = x)$ up to some fixed absolute error ϵ has a randomized polynomial time algorithm. Therefore, the problem cannot be $\mathcal{NP}$-hard unless $\mathcal{NP} = \mathcal{RP}$. This result is an improvement on the exact case, where even the task of computing $P(X = x)$ is $\mathcal{NP}$-hard.

Unfortunately, the good news is very limited in scope, in that it disappears once we introduce evidence. Specifically, it is $\mathcal{NP}$-hard to find an absolute approximation to $P(x \mid \boldsymbol{e})$ for any $\epsilon < 1/2$.

Theorem 9.4

The following problem is $\mathcal{NP}$-hard for any $\epsilon \in (0, 1/2)$:

> *Given a Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a variable $X \in \mathcal{X}$, a value $x \in Val(X)$, and an observation $\boldsymbol{E} = \boldsymbol{e}$ for $\boldsymbol{E} \subset \mathcal{X}$ and $\boldsymbol{e} \in Val(\boldsymbol{E})$, find a number ρ that has absolute error ϵ for $P_{\mathcal{B}}(X = x \mid \boldsymbol{e})$.*

Proof The proof uses the same construction that we used before. Consider a formula ϕ, and consider the analogous BN $\mathcal{B}$, as described in theorem 9.1. Recall that our BN had a variable Q_i for each propositional variable q_i in our Boolean formula, a bunch of other intermediate

variables, and then a variable X whose value, given any assignment of values q_1^1, q_1^0 to the Q_i's, was the associated truth value of the formula. We now show that, given such an approximation algorithm, we can decide whether the formula is satisfiable. We begin by computing $P(Q_1 \mid x^1)$. We pick the value v_1 for Q_1 that is most likely given x^1, and we instantiate it to this value. That is, we generate a network $\mathcal{B}_2$ that does not contain Q_1, and that represents the distribution $\mathcal{B}$ conditioned on $Q_1 = v_1$. We repeat this process for $Q_2, \ldots, Q_n$. This results in some assignment $v_1, \ldots, v_n$ to the Q_i's. We now prove that this is a satisfying assignment if and only if the original formula ϕ was satisfiable.

We begin with the easy case. If ϕ is not satisfiable, then $v_1, \ldots, v_n$ can hardly be a satisfying assignment for it. Now, assume that ϕ is satisfiable. We show that it also has a satisfying assignment with $Q_1 = v_1$. If ϕ is satisfiable with both $Q_1 = q_1^1$ and $Q_1 = q_1^0$, then this is obvious. Assume, however, that ϕ is satisfiable, but not when $Q_1 = v$. Then necessarily, we will have that $P(Q_1 = v \mid x^1)$ is 0, and the probability of the complementary event is 1. If we have an approximation ρ whose error is guaranteed to be $< 1/2$, then choosing the v that maximizes this probability is guaranteed to pick the v whose probability is 1. Thus, in either case the formula has a satisfying assignment where $Q_1 = v$.

We can continue in this fashion, proving by induction on k that ϕ has a satisfying assignment with $Q_1 = v_1, \ldots, Q_k = v_k$. In the case where ϕ is satisfiable, this process will terminate with a satisfying assignment. In the case where ϕ is not, it clearly will not terminate with a satisfying assignment. We can determine which is the case simply by checking whether the resulting assignment satisfies ϕ. This gives us a polynomial time process for deciding satisfiability. ■

Because $\epsilon = 1/2$ corresponds to random guessing, this result is quite discouraging. It tells us that, in the case where we have evidence, approximate inference is no easier than exact inference, in the worst case.

9.2 Variable Elimination: The Basic Ideas

We begin our discussion of inference by discussing the principles underlying exact inference in graphical models. As we show, the same graphical structure that allows a compact representation of complex distributions also help support inference. In particular, we can use dynamic programming techniques (as discussed in appendix A.3.3) to perform inference even for certain large and complex networks in a very reasonable time. We now provide the intuition underlying these algorithms, an intuition that is presented more formally in the remainder of this chapter.

We begin by considering the inference task in a very simple network $A \rightarrow B \rightarrow C \rightarrow D$. We first provide a phased computation, which uses results from the previous phase for the computation in the next phase. We then reformulate this process in terms of a global computation on the joint distribution.

Assume that our first goal is to compute the probability $P(B)$, that is, the distribution over values b of B. Basic probabilistic reasoning (with no assumptions) tells us that

$$P(B) = \sum_a P(a)P(B \mid a). \tag{9.4}$$

Fortunately, we have all the required numbers in our Bayesian network representation: each number $P(a)$ is in the CPD for A, and each number $P(b \mid a)$ is in the CPD for B. Note that

if A has k values and B has m values, the number of basic arithmetic operations required is $O(k \times m)$: to compute $P(b)$, we must multiply $P(b \mid a)$ with $P(a)$ for each of the k values of A, and then add them up, that is, k multiplications and $k-1$ additions; this process must be repeated for each of the m values b.

Now, assume we want to compute $P(C)$. Using the same analysis, we have that

$$P(C) = \sum_b P(b)P(C \mid b). \tag{9.5}$$

Again, the conditional probabilities $P(c \mid b)$ are known: they constitute the CPD for C. The probability of B is not specified as part of the network parameters, but equation (9.4) shows us how it can be computed. Thus, we can compute $P(C)$. We can continue the process in an analogous way, in order to compute $P(D)$.

Note that the structure of the network, and its effect on the parameterization of the CPDs, is critical for our ability to perform this computation as described. Specifically, assume that A had been a parent of C. In this case, the CPD for C would have included A, and our computation of $P(B)$ would not have sufficed for equation (9.5).

Also note that this algorithm does not compute single values, but rather sets of values at a time. In particular equation (9.4) computes an entire distribution over all of the possible values of B. All of these are then used in equation (9.5) to compute $P(C)$. This property turns out to be critical for the performance of the general algorithm.

Let us analyze the complexity of this process on a general chain. Assume that we have a chain with n variables $X_1 \rightarrow \ldots \rightarrow X_n$, where each variable in the chain has k values. As described, the algorithm would compute $P(X_{i+1})$ from $P(X_i)$, for $i = 1, \ldots, n-1$. Each such step would consist of the following computation:

$$P(X_{i+1}) = \sum_{x_i} P(X_{i+1} \mid x_i)P(x_i),$$

where $P(X_i)$ is computed in the previous step. The cost of each such step is $O(k^2)$: The distribution over X_i has k values, and the CPD $P(X_{i+1} \mid X_i)$ has k^2 values; we need to multiply $P(x_i)$, for each value x_i, with each CPD entry $P(x_{i+1} \mid x_i)$ (k^2 multiplications), and then, for each value x_{i+1}, sum up the corresponding entries ($k \times (k-1)$ additions). We need to perform this process for every variable $X_2, \ldots, X_n$; hence, the total cost is $O(nk^2)$.

By comparison, consider the process of generating the entire joint and summing it out, which requires that we generate k^n probabilities for the different events $x_1, \ldots, x_n$. Hence, we have at least one example where, despite the exponential size of the joint distribution, we can do inference in linear time.

Using this process, we have managed to do inference over the joint distribution without ever generating it explicitly. What is the basic insight that allows us to avoid the exhaustive enumeration? Let us reexamine this process in terms of the joint $P(A, B, C, D)$. By the chain rule for Bayesian networks, the joint decomposes as

$$P(A)P(B \mid A)P(C \mid B)P(D \mid C)$$

To compute $P(D)$, we need to sum together all of the entries where $D = d^1$, and to (separately) sum together all of the entries where $D = d^2$. The exact computation that needs to be

$$
\begin{array}{rcccc}
 & P(a^1) & P(b^1 \mid a^1) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & P(a^1) & P(b^1 \mid a^1) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^2 \mid b^2) & P(d^1 \mid c^2)
\end{array}
$$

$$
\begin{array}{rcccc}
 & P(a^1) & P(b^1 \mid a^1) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & P(a^1) & P(b^1 \mid a^1) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^2 \mid b^2) & P(d^2 \mid c^2) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}
$$

Figure 9.2 Computing $P(D)$ by summing over the joint distribution for a chain $A \rightarrow B \rightarrow C \rightarrow D$; all of the variables are binary valued.

performed, for binary-valued variables A, B, C, D, is shown in figure 9.2.[1]

Examining this summation, we see that it has a lot of structure. For example, the third and fourth terms in the first two entries are both $P(c^1 \mid b^1)P(d^1 \mid c^1)$. We can therefore modify the computation to first compute

$$P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)$$

and only then multiply by the common term. The same structure is repeated throughout the table. If we perform the same transformation, we get a new expression, as shown in figure 9.3.

We now observe that certain terms are repeated several times in this expression. Specifically, $P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)$ and $P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)$ are each repeated four times. Thus, it seems clear that we can gain significant computational savings by computing them once and then storing them. There are two such expressions, one for each value of B. Thus, we define a function $\tau_1 : Val(B) \mapsto \mathbb{R}$, where $\tau_1(b^1)$ is the first of these two expressions, and $\tau_1(b^2)$ is the second. Note that $\tau_1(B)$ corresponds exactly to $P(B)$.

The resulting expression, assuming $\tau_1(B)$ has been computed, is shown in figure 9.4. Examining this new expression, we see that we once again can reverse the order of a sum and a product, resulting in the expression of figure 9.5. And, once again, we notice some shared expressions, that are better computed once and used multiple times. We define $\tau_2 : Val(C) \mapsto \mathbb{R}$.

$$
\begin{array}{rcl}
\tau_2(c^1) & = & \tau_1(b^1)P(c^1 \mid b^1) + \tau_1(b^2)P(c^1 \mid b^2) \\
\tau_2(c^2) & = & \tau_1(b^1)P(c^2 \mid b^1) + \tau_1(b^2)P(c^2 \mid b^2)
\end{array}
$$

1. When D is binary-valued, we can get away with doing only the first of these computations. However, this trick does not carry over to the case of variables with more than two values or to the case where we have evidence. Therefore, our example will show the computation in its generality.

$$
\begin{array}{rlll}
 & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
 & & & \\
 & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}
$$

Figure 9.3 The first transformation on the sum of figure 9.2

$$
\begin{array}{rlll}
 & \tau_1(b^1) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & \tau_1(b^2) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & \tau_1(b^1) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & \tau_1(b^2) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
 & & & \\
 & \tau_1(b^1) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & \tau_1(b^2) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & \tau_1(b^1) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & \tau_1(b^2) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}
$$

Figure 9.4 The second transformation on the sum of figure 9.2

$$
\begin{array}{rll}
 & (\tau_1(b^1)P(c^1 \mid b^1) + \tau_1(b^2)P(c^1 \mid b^2)) & P(d^1 \mid c^1) \\
+ & (\tau_1(b^1)P(c^2 \mid b^1) + \tau_1(b^2)P(c^2 \mid b^2)) & P(d^1 \mid c^2) \\
 & & \\
 & (\tau_1(b^1)P(c^1 \mid b^1) + \tau_1(b^2)P(c^1 \mid b^2)) & P(d^2 \mid c^1) \\
+ & (\tau_1(b^1)P(c^2 \mid b^1) + \tau_1(b^2)P(c^2 \mid b^2)) & P(d^2 \mid c^2)
\end{array}
$$

Figure 9.5 The third transformation on the sum of figure 9.2

$$
\begin{array}{rll}
 & \tau_2(c^1) & P(d^1 \mid c^1) \\
+ & \tau_2(c^2) & P(d^1 \mid c^2) \\
 & & \\
 & \tau_2(c^1) & P(d^2 \mid c^1) \\
+ & \tau_2(c^2) & P(d^2 \mid c^2)
\end{array}
$$

Figure 9.6 The fourth transformation on the sum of figure 9.2

The final expression is shown in figure 9.6.

Summarizing, we begin by computing $\tau_1(B)$, which requires four multiplications and two additions. Using it, we can compute $\tau_2(C)$, which also requires four multiplications and two additions. Finally, we can compute $P(D)$, again, at the same cost. The total number of operations is therefore 18. By comparison, generating the joint distribution requires $16 \cdot 3 = 48$

multiplications (three for each of the 16 entries in the joint), and 14 additions (7 for each of $P(d^1)$ and $P(d^2)$).

Written somewhat more compactly, the transformation we have performed takes the following steps: We want to compute

$$P(D) = \sum_C \sum_B \sum_A P(A)P(B \mid A)P(C \mid B)P(D \mid C).$$

We push in the first summation, resulting in

$$\sum_C P(D \mid C) \sum_B P(C \mid B) \sum_A P(A)P(B \mid A).$$

We compute the product $\psi_1(A, B) = P(A)P(B \mid A)$ and then sum out A to obtain the function $\tau_1(B) = \sum_A \psi_1(A, B)$. Specifically, for each value b, we compute $\tau_1(b) = \sum_A \psi_1(A, b) = \sum_A P(A)P(b \mid A)$. We then continue by computing:

$$\begin{aligned} \psi_2(B, C) &= \tau_1(B)P(C \mid B) \\ \tau_2(C) &= \sum_B \psi_2(B, C). \end{aligned}$$

This computation results in a new vector $\tau_2(C)$, which we then proceed to use in the final phase of computing $P(D)$.

dynamic programming

This procedure is performing *dynamic programming* (see appendix A.3.3); doing this summation the naive way would have us compute every $P(b) = \sum_A P(A)P(b \mid A)$ many times, once for every value of C and D. In general, in a chain of length n, this internal summation would be computed exponentially many times. Dynamic programming "inverts" the order of computation — performing it inside out instead of outside in. Specifically, we perform the innermost summation first, computing once and for all the values in $\tau_1(B)$; that allows us to compute $\tau_2(C)$ once and for all, and so on.

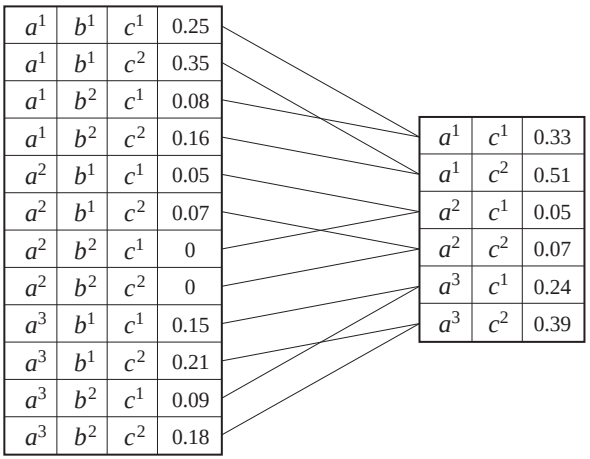
To summarize, the two ideas that help us address the exponential blowup of the joint distribution are:

- **Because of the structure of the Bayesian network, some subexpressions in the joint depend only on a small number of variables.**
- **By computing these expressions once and caching the results, we can avoid generating them exponentially many times.**

9.3 Variable Elimination

factor

To formalize the algorithm demonstrated in the previous section, we need to introduce some basic concepts. In chapter 4, we introduced the notion of a *factor* ϕ over a scope $Scope[\phi] = \boldsymbol{X}$, which is a function $\phi : Val(\boldsymbol{X}) \mapsto \mathbb{R}$. The main steps in the algorithm described here can be viewed as a manipulation of factors. Importantly, by using the factor-based view, we can define the algorithm in a general form that applies equally to Bayesian networks and Markov networks.



a^1	b^1	c^1	0.25
a^1	b^1	c^2	0.35
a^1	b^2	c^1	0.08
a^1	b^2	c^2	0.16
a^2	b^1	c^1	0.05
a^2	b^1	c^2	0.07
a^2	b^2	c^1	0
a^2	b^2	c^2	0
a^3	b^1	c^1	0.15
a^3	b^1	c^2	0.21
a^3	b^2	c^1	0.09
a^3	b^2	c^2	0.18

a^1	c^1	0.33
a^1	c^2	0.51
a^2	c^1	0.05
a^2	c^2	0.07
a^3	c^1	0.24
a^3	c^2	0.39

Figure 9.7 Example of factor marginalization: summing out B.

9.3.1 Basic Elimination

9.3.1.1 Factor Marginalization

The key operation that we are performing when computing the probability of some subset of variables is that of marginalizing out variables from a distribution. That is, we have a distribution over a set of variables $\mathcal{X}$, and we want to compute the marginal of that distribution over some subset $\boldsymbol{X}$. We can view this computation as an operation on a factor:

Definition 9.3 factor marginalization

Let $\boldsymbol{X}$ be a set of variables, and $Y \notin \boldsymbol{X}$ a variable. Let $\phi(\boldsymbol{X}, Y)$ be a factor. We define the factor marginalization *of Y in ϕ, denoted $\sum_Y \phi$, to be a factor ψ over $\boldsymbol{X}$ such that:*

$$\psi(\boldsymbol{X}) = \sum_Y \phi(\boldsymbol{X}, Y).$$

This operation is also called summing out *of Y in ψ.* ■

The key point in this definition is that we only sum up entries in the table where the values of $\boldsymbol{X}$ match up. Figure 9.7 illustrates this process.

The process of marginalizing a joint distribution $P(\boldsymbol{X}, \boldsymbol{Y})$ onto $\boldsymbol{X}$ in a Bayesian network is simply summing out the variables $\boldsymbol{Y}$ in the factor corresponding to P. If we sum out all variables, we get a factor consisting of a single number whose value is 1. If we sum out all of the variables in the unnormalized distribution $\tilde{P}_\Phi$ defined by the product of factors in a Markov network, we get the partition function.

A key observation used in performing inference in graphical models is that the operations of factor product and summation behave precisely as do product and summation over numbers. Specifically, both operations are commutative, so that $\phi_1 \cdot \phi_2 = \phi_2 \cdot \phi_1$ and $\sum_X \sum_Y \phi = \sum_Y \sum_X \phi$. Products are also associative, so that $(\phi_1 \cdot \phi_2) \cdot \phi_3 = \phi_1 \cdot (\phi_2 \cdot \phi_3)$. Most importantly,

Algorithm 9.1 Sum-product variable elimination algorithm

```
Procedure Sum-Product-VE (
    Φ,    // Set of factors
    Z,    // Set of variables to be eliminated
    ≺     // Ordering on Z
)
1   Let Z_1, ..., Z_k be an ordering of Z such that
2     Z_i ≺ Z_j if and only if i < j
3   for i = 1, ..., k
4     Φ ← Sum-Product-Eliminate-Var(Φ, Z_i)
5   φ* ← ∏_{φ∈Φ} φ
6   return φ*

Procedure Sum-Product-Eliminate-Var (
    Φ,    // Set of factors
    Z     // Variable to be eliminated
)
1   Φ' ← {φ ∈ Φ : Z ∈ Scope[φ]}
2   Φ'' ← Φ − Φ'
3   ψ ← ∏_{φ∈Φ'} φ
4   τ ← ∑_Z ψ
5   return Φ'' ∪ {τ}
```

we have a simple rule allowing us to exchange summation and product: If $X \notin \mathit{Scope}[\phi_1]$, then

$$\sum_X (\phi_1 \cdot \phi_2) = \phi_1 \cdot \sum_X \phi_2. \tag{9.6}$$

9.3.1.2 The Variable Elimination Algorithm

The key to both of our examples in the last section is the application of equation (9.6). Specifically, in our chain example of section 9.2, we can write:

$$P(A, B, C, D) = \phi_A \cdot \phi_B \cdot \phi_C \cdot \phi_D.$$

On the other hand, the marginal distribution over D is

$$P(D) = \sum_C \sum_B \sum_A P(A, B, C, D).$$

Applying equation (9.6), we can now conclude:

$$\begin{aligned} P(D) &= \sum_C \sum_B \sum_A \phi_A \cdot \phi_B \cdot \phi_C \cdot \phi_D \\ &= \sum_C \sum_B \phi_C \cdot \phi_D \cdot \left(\sum_A \phi_A \cdot \phi_B \right) \\ &= \sum_C \phi_D \cdot \left(\sum_B \phi_C \cdot \left(\sum_A \phi_A \cdot \phi_B \right) \right), \end{aligned}$$

where the different transformations are justified by the limited scope of the CPD factors; for example, the second equality is justified by the fact that the scope of ϕ_C and ϕ_D does not contain A. In general, any marginal probability computation involves taking the product of all the CPDs, and doing a summation on all the variables except the query variables. We can do these steps in any order we want, as long as we only do a summation on a variable X *after* multiplying in all of the factors that involve X.

In general, we can view the task at hand as that of computing the value of an expression of the form:

$$\sum_{\boldsymbol{Z}} \prod_{\phi \in \Phi} \phi.$$

sum-product

We call this task the *sum-product* inference task. The key insight that allows the effective computation of this expression is the fact that the scope of the factors is limited, allowing us to "push in" some of the summations, performing them over the product of only a subset of factors. One simple instantiation of this algorithm is a procedure called *sum-product variable elimination* (VE), shown in algorithm 9.1. The basic idea in the algorithm is that we sum out variables one at a time. When we sum out any variable, we multiply all the factors that mention that variable, generating a product factor. Now, we sum out the variable from this combined factor, generating a new factor that we enter into our set of factors to be dealt with.

variable elimination

Based on equation (9.6), the following result follows easily:

Theorem 9.5

Let $\boldsymbol{X}$ be some set of variables, and let Φ be a set of factors such that for each $\phi \in \Phi$, Scope$[\phi] \subseteq \boldsymbol{X}$. Let $\boldsymbol{Y} \subset \boldsymbol{X}$ be a set of query variables, and let $\boldsymbol{Z} = \boldsymbol{X} - \boldsymbol{Y}$. Then for any ordering $\prec$ over $\boldsymbol{Z}$, Sum-Product-VE$(\Phi, \boldsymbol{Z}, \prec)$ *returns a factor $\phi^*(\boldsymbol{Y})$ such that*

$$\phi^*(\boldsymbol{Y}) = \sum_{\boldsymbol{Z}} \prod_{\phi \in \Phi} \phi.$$

We can apply this algorithm to the task of computing the probability distribution $P_{\mathcal{B}}(\boldsymbol{Y})$ for a Bayesian network $\mathcal{B}$. We simply instantiate Φ to consist of all of the CPDs:

$$\Phi = \{\phi_{X_i}\}_{i=1}^n$$

where $\phi_{X_i} = P(X_i \mid \mathrm{Pa}_{X_i})$. We then apply the variable elimination algorithm to the set $\{Z_1, \ldots, Z_m\} = \mathcal{X} - \boldsymbol{Y}$ (that is, we eliminate all the nonquery variables).

We can also apply precisely the same algorithm to the task of computing conditional probabilities in a Markov network. We simply initialize the factors to be the clique potentials and

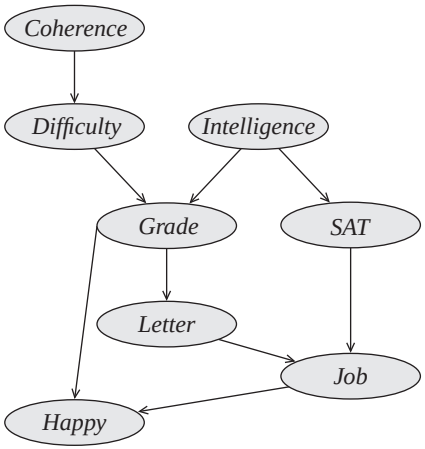


Figure 9.8 The Extended-Student **Bayesian network**

run the elimination algorithm. As for Bayesian networks, we then apply the variable elimination algorithm to the set $\boldsymbol{Z} = \mathcal{X} - \boldsymbol{Y}$. The procedure returns an *unnormalized* factor over the query variables $\boldsymbol{Y}$. The distribution over $\boldsymbol{Y}$ can be obtained by normalizing the factor; the partition function is simply the normalizing constant.

Example 9.1

Let us demonstrate the procedure on a nontrivial example. Consider the network demonstrated in figure 9.8, which is an extension of our Student *network. The chain rule for this network asserts that*

$$\begin{aligned} P(C,D,I,G,S,L,J,H) &= P(C)P(D \mid C)P(I)P(G \mid I,D)P(S \mid I) \\ &\quad P(L \mid G)P(J \mid L,S)P(H \mid G,J) \\ &= \phi_C(C)\phi_D(D,C)\phi_I(I)\phi_G(G,I,D)\phi_S(S,I) \\ &\quad \phi_L(L,G)\phi_J(J,L,S)\phi_H(H,G,J). \end{aligned}$$

We will now apply the VE algorithm to compute $P(J)$. We will use the elimination ordering: C, D, I, H, G, S, L:

1. *Eliminating C: We compute the factors*

$$\begin{aligned} \psi_1(C,D) &= \phi_C(C) \cdot \phi_D(D,C) \\ \tau_1(D) &= \sum_C \psi_1. \end{aligned}$$

2. *Eliminating D: Note that we have already eliminated one of the original factors that involve D — $\phi_D(D,C) = P(D \mid C)$. On the other hand, we introduced the factor $\tau_1(D)$ that involves*

D. Hence, we now compute:

$$\begin{aligned}\psi_2(G, I, D) &= \phi_G(G, I, D) \cdot \tau_1(D) \\ \tau_2(G, I) &= \sum_D \psi_2(G, I, D).\end{aligned}$$

3. Eliminating I: We compute the factors

$$\begin{aligned}\psi_3(G, I, S) &= \phi_I(I) \cdot \phi_S(S, I) \cdot \tau_2(G, I) \\ \tau_3(G, S) &= \sum_I \psi_3(G, I, S).\end{aligned}$$

4. Eliminating H: We compute the factors

$$\begin{aligned}\psi_4(G, J, H) &= \phi_H(H, G, J) \\ \tau_4(G, J) &= \sum_H \psi_4(G, J, H).\end{aligned}$$

Note that $\tau_4 \equiv 1$ (all of its entries are exactly 1): we are simply computing $\sum_H P(H \mid G, J)$, which is a probability distribution for every G, J, and hence sums to 1. A naive execution of this algorithm will end up generating this factor, which has no value. Generating it has no impact on the final answer, but it does complicate the algorithm. In particular, the existence of this factor complicates our computation in the next step.

5. Eliminating G: We compute the factors

$$\begin{aligned}\psi_5(G, J, L, S) &= \tau_4(G, J) \cdot \tau_3(G, S) \cdot \phi_L(L, G) \\ \tau_5(J, L, S) &= \sum_G \psi_5(G, J, L, S).\end{aligned}$$

Note that, without the factor $\tau_4(G, J)$, the results of this step would not have involved J.

6. Eliminating S: We compute the factors

$$\begin{aligned}\psi_6(J, L, S) &= \tau_5(J, L, S) \cdot \phi_J(J, L, S) \\ \tau_6(J, L) &= \sum_S \psi_6(J, L, S).\end{aligned}$$

7. Eliminating L: We compute the factors

$$\begin{aligned}\psi_7(J, L) &= \tau_6(J, L) \\ \tau_7(J) &= \sum_L \psi_7(J, L).\end{aligned}$$

We summarize these steps in table 9.1.

Note that we can use any elimination ordering. For example, consider eliminating variables in the order G, I, S, L, H, C, D. We would then get the behavior of table 9.2. The result, as before, is precisely $P(J)$. However, note that this elimination ordering introduces factors with much larger scope. We return to this point later on. ■

Step	Variable eliminated	Factors used	Variables involved	New factor
1	C	$\phi_C(C)$, $\phi_D(D, C)$	C, D	$\tau_1(D)$
2	D	$\phi_G(G, I, D)$, $\tau_1(D)$	G, I, D	$\tau_2(G, I)$
3	I	$\phi_I(I)$, $\phi_S(S, I)$, $\tau_2(G, I)$	G, S, I	$\tau_3(G, S)$
4	H	$\phi_H(H, G, J)$	H, G, J	$\tau_4(G, J)$
5	G	$\tau_4(G, J)$, $\tau_3(G, S)$, $\phi_L(L, G)$	G, J, L, S	$\tau_5(J, L, S)$
6	S	$\tau_5(J, L, S)$, $\phi_J(J, L, S)$	J, L, S	$\tau_6(J, L)$
7	L	$\tau_6(J, L)$	J, L	$\tau_7(J)$

Table 9.1 A run of variable elimination for the query $P(J)$

Step	Variable eliminated	Factors used	Variables involved	New factor
1	G	$\phi_G(G, I, D)$, $\phi_L(L, G)$, $\phi_H(H, G, J)$	G, I, D, L, J, H	$\tau_1(I, D, L, J, H)$
2	I	$\phi_I(I)$, $\phi_S(S, I)$, $\tau_1(I, D, L, S, J, H)$	S, I, D, L, J, H	$\tau_2(D, L, S, J, H)$
3	S	$\phi_J(J, L, S)$, $\tau_2(D, L, S, J, H)$	D, L, S, J, H	$\tau_3(D, L, J, H)$
4	L	$\tau_3(D, L, J, H)$	D, L, J, H	$\tau_4(D, J, H)$
5	H	$\tau_4(D, J, H)$	D, J, H	$\tau_5(D, J)$
6	C	$\phi_C(C)$, $\phi_D(D, C)$	D, J, C	$\tau_6(D)$
7	D	$\tau_5(D, J)$, $\tau_6(D)$	D, J	$\tau_7(J)$

Table 9.2 A different run of variable elimination for the query $P(J)$

9.3.1.3 Semantics of Factors

It is interesting to consider the semantics of the intermediate factors generated as part of this computation. In many of the examples we have given, they correspond to marginal or conditional probabilities in the network. However, although these factors often correspond to such probabilities, this is not always the case. Consider, for example, the network of figure 9.9a. The result of eliminating the variable X is a factor

$$\tau(A, B, C) = \sum_X P(X) \cdot P(A \mid X) \cdot P(C \mid B, X).$$

This factor does not correspond to any probability or conditional probability in this network. To understand why, consider the various options for the meaning of this factor. Clearly, it cannot be a conditional distribution where B is on the left hand side of the conditioning bar (for example, $P(A, B, C)$), as $P(B \mid A)$ has not yet been multiplied in. The most obvious candidate is $P(A, C \mid B)$. However, this conjecture is also false. The probability $P(A \mid B)$ relies heavily on the properties of the CPD $P(B \mid A)$; for example, if B is deterministically equal to A, $P(A \mid B)$ has a very different form than if B depends only very weakly on A. Since the CPD $P(B \mid A)$ was not taken into consideration when computing $\tau(A, B, C)$, it cannot represent the conditional probability $P(A, C \mid B)$. In general, we can verify that this factor

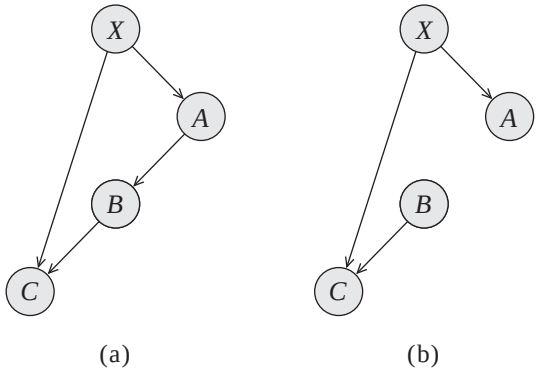


Figure 9.9 Understanding intermediate factors in variable elimination as conditional probabilities: (a) A Bayesian network where elimination does not lead to factors that have an interpretation as conditional probabilities. (b) A different Bayesian network where the resulting factor does correspond to a conditional probability.

does not correspond to any conditional probability expression in this network.

It is interesting to note, however, that the resulting factor does, in fact, correspond to a conditional probability $P(A, C \mid B)$, but *in a different network*: the one shown in figure 9.9b, where all CPDs except for B are the same. In fact, this phenomenon is a general one (see exercise 9.2).

9.3.2 Dealing with Evidence

It remains only to consider how we would introduce evidence. For example, assume we observe the value i^1 (the student is intelligent) and h^0 (the student is unhappy). Our goal is to compute $P(J \mid i^1, h^0)$. First, we reduce this problem to computing the unnormalized distribution $P(J, i^1, h^0)$. From this intermediate result, we can compute the conditional probability as in equation (9.1), by renormalizing by the probability of the evidence $P(i^1, h^0)$.

How do we compute $P(J, i^1, h^0)$? The key observation is proposition 4.7, which shows us how to view, as a Gibbs distribution, an unnormalized measure derived from introducing evidence into a Bayesian network. Thus, we can view this computation as summing out all of the entries in the *reduced factor*: $P[i^1 h^0]$ whose scope is $\{C, D, G, L, S, J\}$. This factor is no longer normalized, but it is still a valid factor.

factor reduction

Based on this observation, we can now apply precisely the same sum-product variable elimination algorithm to the task of computing $P(\boldsymbol{Y}, \boldsymbol{e})$. We simply apply the algorithm to the set of factors in the network, reduced by $\boldsymbol{E} = \boldsymbol{e}$, and eliminate the variables in $\mathcal{X} - \boldsymbol{Y} - \boldsymbol{E}$. The returned factor $\phi^*(\boldsymbol{Y})$ is precisely $P(\boldsymbol{Y}, \boldsymbol{e})$. To obtain $P(\boldsymbol{Y} \mid \boldsymbol{e})$ we simply renormalize $\phi^*(\boldsymbol{Y})$ by multiplying it by $\frac{1}{\alpha}$ to obtain a legal distribution, where α is the sum over the entries in our unnormalized distribution, which represents the probability of the evidence. To summarize, the algorithm for computing conditional probabilities in a Bayesian or Markov network is shown in algorithm 9.2.

We demonstrate this process on the example of computing $P(J, i^1, h^0)$. We use the same

Algorithm 9.2 Using Sum-Product-VE **for computing conditional probabilities**

Procedure Cond-Prob-VE (
  $\mathcal{K}$, // A network over $\mathcal{X}$
  $\boldsymbol{Y}$, // Set of query variables
  $\boldsymbol{E} = \boldsymbol{e}$ // Evidence
)
1 $\Phi \leftarrow$ Factors parameterizing $\mathcal{K}$
2 Replace each $\phi \in \Phi$ by $\phi[\boldsymbol{E} = \boldsymbol{e}]$
3 Select an elimination ordering $\prec$
4 $\boldsymbol{Z} \leftarrow = \mathcal{X} - \boldsymbol{Y} - \boldsymbol{E}$
5 $\phi^* \leftarrow$ Sum-Product-VE($\Phi, \prec, \boldsymbol{Z}$)
6 $\alpha \leftarrow \sum_{\boldsymbol{y} \in Val(\boldsymbol{Y})} \phi^*(\boldsymbol{y})$
7 **return** α, ϕ^*

Step	Variable eliminated	Factors used	Variables involved	New factor
1'	C	$\phi_C(C)$, $\phi_D(D, C)$	C, D	$\tau'_1(D)$
2'	D	$\phi_G[I = i^1](G, D)$, $\phi_I[I = i^1]()$, $\tau'_1(D)$	G, D	$\tau'_2(G)$
5'	G	$\tau'_2(G)$, $\phi_L(L, G)$, $\phi_H[H = h^0](G, J)$	G, L, J	$\tau'_5(L, J)$
6'	S	$\phi_S[I = i^1](S)$, $\phi_J(J, L, S)$	J, L, S	$\tau'_6(J, L)$
7'	L	$\tau'_6(J, L)$, $\tau'_5(J, L)$	J, L	$\tau'_7(J)$

Table 9.3 A run of sum-product variable elimination for $P(J, i^1, h^0)$

elimination ordering that we used in table 9.1. The results are shown in table 9.3; the step numbers correspond to the steps in table 9.1. It is interesting to note the differences between the two runs of the algorithm. First, we notice that steps (3) and (4) disappear in the computation with evidence, since I and H do not need to be eliminated. More interestingly, by not eliminating I, we avoid the step that correlates G and S. In this execution, G and S never appear together in the same factor; they are both eliminated, and only their end results are combined. Intuitively, G and S are conditionally independent given I; hence, observing I renders them independent, so that we do not have to consider their joint distribution explicitly. Finally, we notice that $\phi_I[I = i^1] = P(i^1)$ is a factor over an empty scope, which is simply a number. It can be multiplied into any factor at any point in the computation. We chose arbitrarily to incorporate it into step (2′). Note that if our goal is to compute a conditional probability given the evidence, and not the probability of the evidence itself, we can avoid multiplying in this factor entirely, since its effect will disappear in the renormalization step at the end.

network polynomial

Box 9.A — Concept: The Network Polynomial. *The* network polynomial *provides an interesting and useful alternative view of variable elimination. We begin with describing the concept for the case of a Gibbs distribution parameterized via a set of full table factors* Φ. *The polynomial* f_Φ

is defined over the following set of variables:

- *For each factor $\phi_c \in \Phi$ with scope $\boldsymbol{X}_c$, we have a variable $\theta_{\boldsymbol{x}_c}$ for every $\boldsymbol{x}_c \in Val(\boldsymbol{X}_c)$.*
- *For each variable X_i and every value $x_i \in Val(X_i)$, we have a binary-valued variable λ_{x_i}.*

In other words, the polynomial has one argument for each of the network parameters and for each possible assignment to a network variable. The polynomial f_Φ is now defined as follows:

$$f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \sum_{x_1,\ldots,x_n} \left(\prod_{\phi_c \in \Phi} \theta_{\boldsymbol{x}_c} \cdot \prod_{i=1}^{n} \lambda_{x_i} \right). \tag{9.7}$$

Evaluating the network polynomial is equivalent to the inference task. In particular, let $\boldsymbol{Y} = \boldsymbol{y}$ be an assignment to some subset of network variables; define an assignment $\boldsymbol{\lambda}^{\boldsymbol{y}}$ as follows:

- *for each $Y_i \in \boldsymbol{Y}$, define $\lambda^{\boldsymbol{y}}_{y_i} = 1$ and $\lambda^{\boldsymbol{y}}_{y'_i} = 0$ for all $y'_i \neq y_i$;*
- *for each $Y_i \notin \boldsymbol{Y}$, define $\lambda^{\boldsymbol{y}}_{y_i} = 1$ for all $y_i \in Val(Y_i)$.*

With this definition, we can now show (exercise 9.4a) that:

$$f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}^{\boldsymbol{y}}) = \tilde{P}_\Phi(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{\theta}). \tag{9.8}$$

The derivatives of the network polynomial are also of significant interest. We can show (exercise 9.4b) that

$$\frac{\partial f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}^{\boldsymbol{y}})}{\partial \lambda_{x_i}} = \tilde{P}_\Phi(x_i, \boldsymbol{y}_{-i} \mid \boldsymbol{\theta}), \tag{9.9}$$

where $\boldsymbol{y}_{-i}$ is the assignment in $\boldsymbol{y}$ to all variables other than X_i. We can also show that

$$\frac{\partial f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}^{\boldsymbol{y}})}{\partial \theta_{\boldsymbol{x}_c}} = \frac{\tilde{P}_\Phi(\boldsymbol{y}, \boldsymbol{x}_c \mid \boldsymbol{\theta})}{\theta_{\boldsymbol{x}_c}}\ ; \tag{9.10}$$

this fact is proved in lemma 19.1. These derivatives can be used for various purposes, including retracting or modifying evidence in the network (exercise 9.4c), and sensitivity analysis *— computing the effect of changes in a network parameter on the answer to a particular probabilistic query (exercise 9.5).*

sensitivity analysis

Of course, as defined, the representation of the network polynomial is exponentially large in the number of variables in the network. However, we can use the algebraic operations performed in a run of variable elimination to define a network polynomial that has precisely the same complexity as the VE run. More interesting, we can also use the same structure to compute efficiently all of the derivatives of the network polynomial, relative both to the λ_i and the $\theta_{\boldsymbol{x}_c}$ (see exercise 9.6).

9.4 Complexity and Graph Structure: Variable Elimination

From the examples we have seen, it is clear that the VE algorithm can be computationally much more efficient than a full enumeration of the joint. In this section, we analyze the complexity of the algorithm, and understand the source of the computational gains.

We also note that, aside from the asymptotic analysis, a careful implementation of this algorithm can have significant ramifications on performance; see box 10.A.

9.4.1 Simple Analysis

Let us begin with a simple analysis of the basic computational operations taken by algorithm 9.1. Assume we have n random variables, and m initial factors; in a Bayesian network, we have $m = n$; in a Markov network, we may have more factors than variables. For simplicity, assume we run the algorithm until all variables are eliminated.

The algorithm consists of a set of elimination steps, where, in each step, the algorithm picks a variable X_i, then multiplies all factors involving that variable. The result is a single large factor ψ_i. The variable then gets summed out of ψ_i, resulting in a new factor τ_i whose scope is the scope of ψ_i minus X_i. Thus, the work revolves around these factors that get created and processed. Let N_i be the number of entries in the factor ψ_i, and let $N_{\max} = \max_i N_i$.

We begin by counting the number of multiplication steps. Here, we note that the total number of factors ever entered into the set of factors Φ is $m + n$: the m initial factors, plus the n factors τ_i. Each of these factors ϕ is multiplied exactly once: when it is multiplied in line 3 of Sum-Product-Eliminate-Var to produce a large factor ψ_i, it is also extracted from Φ. The cost of multiplying ϕ to produce ψ_i is at most N_i, since each entry of ϕ is multiplied into exactly one entry of ψ_i. Thus, the total number of multiplication steps is at most $(n+m)N_i \leq (n+m)N_{\max} = O(mN_{\max})$. To analyze the number of addition steps, we note that the marginalization operation in line 4 touches each entry in ψ_i exactly once. Thus, the cost of this operation is exactly N_i; we execute this operation once for each factor ψ_i, so that the total number of additions is at most $nN_{\max}$. Overall, the total amount of work required is $O(mN_{\max})$.

The source of the inevitable exponential blowup is the potentially exponential size of the factors ψ_i. If each variable has no more than v values, and a factor ψ_i has a scope that contains k_i variables, then $N_i \leq v^{k_i}$. Thus, we see that the computational cost of the VE algorithm is dominated by the sizes of the intermediate factors generated, with an exponential growth in the number of variables in a factor.

9.4.2 Graph-Theoretic Analysis

Although the size of the factors created during the algorithm is clearly the dominant quantity in the complexity of the algorithm, it is not clear how it relates to the properties of our problem instance. In our case, the only aspect of the problem instance that affects the complexity of the algorithm is the structure of the underlying graph that induced the set of factors on which the algorithm was run. In this section, we reformulate our complexity analysis in terms of this graph structure.

9.4.2.1 Factors and Undirected Graphs

We begin with the observation that the algorithm does not care whether the graph that generated the factors is directed, undirected, or partly directed. The algorithm's input is a set of factors Φ, and the only relevant aspect to the computation is the scope of the factors. Thus, it is easiest to view the algorithm as operating on an undirected graph $\mathcal{H}$.

More precisely, we can define the notion of an undirected graph associated with a set of factors:

Definition 9.4

Let Φ be a set of factors. We define

$$\mathit{Scope}[\Phi] = \cup_{\phi \in \Phi} \mathit{Scope}[\phi]$$

to be the set of all variables appearing in any of the factors in Φ. We define $\mathcal{H}_\Phi$ to be the undirected graph whose nodes correspond to the variables in $\mathit{Scope}[\Phi]$ and where we have an edge X_i—$X_j \in \mathcal{H}_\Phi$ if and only if there exists a factor $\phi \in \Phi$ such that $X_i, X_j \in \mathit{Scope}[\phi]$. ■

In words, the undirected graph $\mathcal{H}_\Phi$ introduces a fully connected subgraph over the scope of each factor $\phi \in \Phi$, and hence is the minimal I-map for the distribution induced by Φ.

We can now show that:

Proposition 9.1 *Let P be a distribution defined by multiplying the factors in Φ and normalizing to define a distribution. Letting $\boldsymbol{X} = \mathit{Scope}[\Phi]$,*

$$P(\boldsymbol{X}) = \frac{1}{Z} \prod_{\phi \in \Phi} \phi,$$

where $Z = \sum_{\boldsymbol{X}} \prod_{\phi \in \Phi} \phi$. Then $\mathcal{H}_\Phi$ is the minimal Markov network I-map for P, and the factors Φ are a parameterization of this network that defines the distribution P.

The proof is left as an exercise (exercise 9.7).

Note that, for a set of factors Φ defined by a Bayesian network $\mathcal{G}$, in the case without evidence, the undirected graph $\mathcal{H}_\Phi$ is precisely the moralized graph of $\mathcal{G}$. In this case, the product of the factors is a normalized distribution, so the partition function of the resulting Markov network is simply 1. Figure 4.6a shows the initial graph for our Student example.

More interesting is the Markov network induced by a set of factors $\Phi[\boldsymbol{e}]$ defined by the reduction of the factors in a Bayesian network to some context $\boldsymbol{E} = \boldsymbol{e}$. In this case, recall that the variables in $\boldsymbol{E}$ are removed from the factors, so $\boldsymbol{X} = \mathit{Scope}[\Phi_{\boldsymbol{e}}] = \mathcal{X} - \boldsymbol{E}$. Furthermore, as we discussed, the unnormalized product of the factors is $P(\boldsymbol{X}, \boldsymbol{e})$, and the partition function of the resulting Markov network is precisely $P(\boldsymbol{e})$. Figure 4.6b shows the initial graph for our Student example with evidence $G = g$, and figure 4.6c shows the case with evidence $G = g, S = s$.

9.4.2.2 Elimination as Graph Transformation

Now, consider the effect of a variable elimination step on the set of factors maintained by the algorithm and on the associated Markov network. When a variable X is eliminated, several operations take place. First, we create a single factor ψ that contains X and all of the variables $\boldsymbol{Y}$ with which it appears in factors. Then, we eliminate X from ψ, replacing it with a new factor τ that contains all of the variables $\boldsymbol{Y}$ but does not contain X. Let Φ_X be the resulting set of factors.

How does the graph $\mathcal{H}_{\Phi_X}$ differ from $\mathcal{H}_\Phi$? The step of constructing ψ generates edges between all of the variables $Y \in \boldsymbol{Y}$. Some of them were present in $\mathcal{H}_\Phi$, whereas others are introduced due to the elimination step; edges that are introduced by an elimination step are called *fill edges*. The step of eliminating X from ψ to construct τ has the effect of removing X and all of its incident edges from the graph.

fill edge

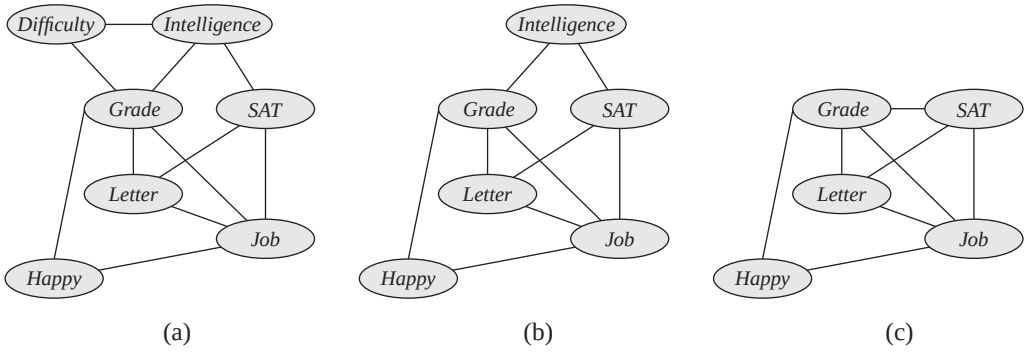


Figure 9.10 Variable elimination as graph transformation in the Student **example,** using the elimination order of table 9.1: (a) after eliminating C; (b) after eliminating D; (c) after eliminating I.

Consider again our Student network, in the case without evidence. As we said, figure 4.6a shows the original Markov network. Figure 9.10a shows the result of eliminating the variable C. Note that there are no fill edges introduced in this step.

After an elimination step, the subsequent elimination steps use the new set of factors. In other words, they can be seen as operations over the new graph. Figure 9.10b and c show the graphs resulting from eliminating first D and then I. Note that the step of eliminating I results in a (new) fill edge G—S, induced by the factor G, I, S.

The computational steps of the algorithm are reflected in this series of graphs. Every factor that appears in one of the steps in the algorithm is reflected in the graph as a clique. In fact, we can summarize the computational cost using a single graph structure.

9.4.2.3 The Induced Graph

We define an undirected graph that is the union of all of the graphs resulting from the different steps of the variable elimination algorithm.

Definition 9.5 induced graph

Let Φ be a set of factors over $\mathcal{X} = \{X_1, \ldots, X_n\}$, and $\prec$ be an elimination ordering for some subset $\boldsymbol{X} \subseteq \mathcal{X}$. The induced graph $\mathcal{I}_{\Phi,\prec}$ *is an undirected graph over $\mathcal{X}$, where X_i and X_j are connected by an edge if they both appear in some intermediate factor ψ generated by the VE algorithm using $\prec$ as an elimination ordering.* ■

For a Bayesian network graph $\mathcal{G}$, we use $\mathcal{I}_{\mathcal{G},\prec}$ to denote the induced graph for the factors Φ corresponding to the CPDs in $\mathcal{G}$; similarly, for a Markov network $\mathcal{H}$, we use $\mathcal{I}_{\mathcal{H},\prec}$ to denote the induced graph for the factors Φ corresponding to the potentials in $\mathcal{H}$.

The induced graph $\mathcal{I}_{\mathcal{G},\prec}$ for our Student example is shown in figure 9.11a. We can see that the fill edge G—S, introduced in step (3) when we eliminated I, is the only fill edge introduced.

As we discussed, each factor ψ used in the computation corresponds to a complete subgraph of the graph $\mathcal{I}_{\mathcal{G},\prec}$ and is therefore a clique in the graph. The connection between cliques in $\mathcal{I}_{\mathcal{G},\prec}$ and factors ψ is, in fact, much tighter:

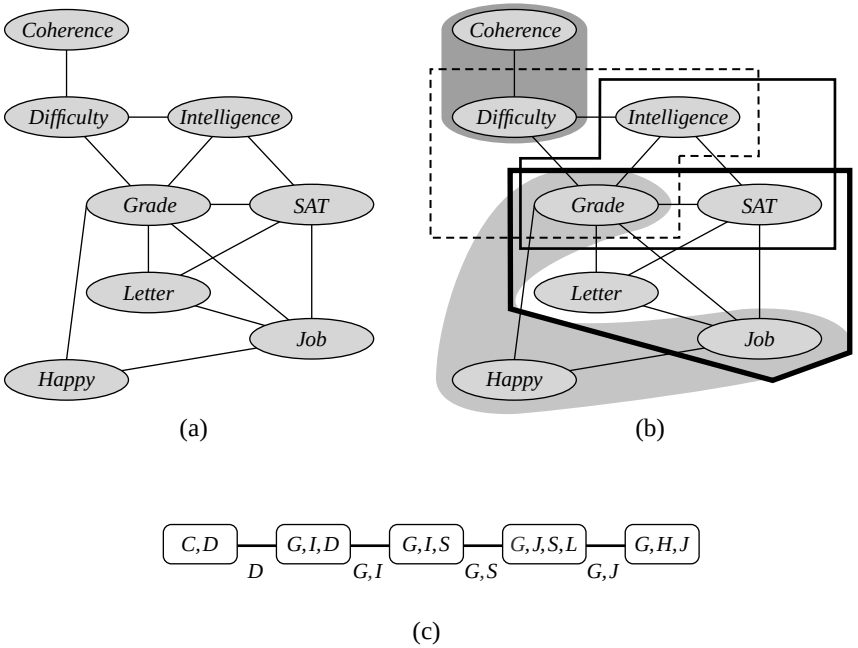


Figure 9.11 Induced graph and clique tree for the Student **example.** (a) Induced graph for variable elimination in the Student example, using the elimination order of table 9.1. (b) Cliques in the induced graph: $\{C, D\}$, $\{D, I, G\}$, $\{G, I, S\}$, $\{G, J, S, L\}$, and $\{G, H, J\}$. (c) Clique tree for the induced graph.

Theorem 9.6

Let $\mathcal{I}_{\Phi,\prec}$ be the induced graph for a set of factors Φ and some elimination ordering $\prec$. Then:

1. *The scope of every factor generated during the variable elimination process is a clique in $\mathcal{I}_{\Phi,\prec}$.*
2. *Every maximal clique in $\mathcal{I}_{\Phi,\prec}$ is the scope of some intermediate factor in the computation.*

PROOF We begin with the first statement. Consider a factor $\psi(Y_1, \ldots, Y_k)$ generated during the VE process. By the definition of the induced graph, there must be an edge between each Y_i and Y_j. Hence $Y_1, \ldots, Y_k$ form a clique.

To prove the second statement, consider some maximal clique $\boldsymbol{Y} = \{Y_1, \ldots, Y_k\}$. Assume, without loss of generality, that Y_1 is the first of the variables in $\boldsymbol{Y}$ in the ordering $\prec$, and is therefore the first among this set to be eliminated. Since $\boldsymbol{Y}$ is a clique, there is an edge from Y_1 to each other Y_i. Note that, once Y_1 is eliminated, it can appear in no more factors, so there can be no new edges added to it. Hence, the edges involving Y_1 were added prior to this point in the computation. The existence of an edge between Y_1 and Y_i therefore implies that, at this point, there is a factor containing both Y_1 and Y_i. When Y_1 is eliminated, all these factors must be multiplied. Therefore, the product step results in a factor ψ that contains all of $Y_1, Y_2, \ldots, Y_k$. Note that this factor can contain no other variables; if it did, these variables would also have an edge to all of $Y_1, \ldots, Y_k$, so that $Y_1, \ldots, Y_k$ would not constitute a maximal connected subgraph. ■

Let us verify that the second property holds for our example. Figure 9.11b shows the maximal cliques in $\mathcal{I}_{\mathcal{G},\prec}$:

$$\begin{aligned}
\boldsymbol{C}_1 &= \{C, D\} \\
\boldsymbol{C}_2 &= \{D, I, G\} \\
\boldsymbol{C}_3 &= \{I, G, S\} \\
\boldsymbol{C}_4 &= \{G, J, L, S\} \\
\boldsymbol{C}_5 &= \{G, H, J\}.
\end{aligned}$$

Both these properties hold for this set of cliques. For example, $\boldsymbol{C}_3$ corresponds to the factor ψ generated in step (5).

☞ **Thus, there is a direct correspondence between the maximal factors generated by our algorithm and maximal cliques in the induced graph. Importantly, the induced graph and the size of the maximal cliques within it depend strongly on the elimination ordering.** Consider, for example, our other elimination ordering for the Student network. In this case, we can verify that our induced graph has a maximal clique over G, I, D, L, J, H, a second over S, I, D, L, J, H, and a third over C, D, J; indeed, the graph is missing only the edge between S and G, and some edges involving C. In this case, the largest clique contains six variables, as opposed to four in our original ordering. Therefore, the cost of computation here is substantially more expensive.

Definition 9.6 *We define the* width *of an induced graph to be the number of nodes in the largest clique in the graph minus 1. We define the* induced width $w_{\mathcal{K},\prec}$ *of an ordering $\prec$ relative to a graph $\mathcal{K}$ (directed or undirected) to be the width of the graph $\mathcal{I}_{\mathcal{K},\prec}$ induced by applying VE to $\mathcal{K}$ using the ordering $\prec$.*

induced width

tree-width

We define the tree-width *of a graph $\mathcal{K}$ to be its minimal induced width $w^*_{\mathcal{K}} = \min_{\prec} w(\mathcal{I}_{\mathcal{K},\prec})$.* ■

The minimal induced width of the graph $\mathcal{K}$ provides us a bound on the best performance we can hope for by applying VE to a probabilistic model that factorizes over $\mathcal{K}$.

9.4.3 Finding Elimination Orderings ⋆

How can we compute the minimal induced width of the graph, and the elimination ordering achieving that width? Unfortunately, there is no easy way to answer this question.

Theorem 9.7 *The following decision problem is $\mathcal{NP}$-complete:*

> *Given a graph $\mathcal{H}$ and some bound K, determine whether there exists an elimination ordering achieving an induced width $\leq K$.*

It follows directly that finding the optimal elimination ordering is also $\mathcal{NP}$-hard. Thus, we cannot easily tell by looking at a graph how computationally expensive inference on it will be. Note that this $\mathcal{NP}$-completeness result is distinct from the $\mathcal{NP}$-hardness of inference itself. That is, even if some oracle gives us the best elimination ordering, the induced width might still be large, and the inference task using that ordering can still require exponential time.

However, as usual, $\mathcal{NP}$-hardness is not the end of the story. There are several techniques that one can use to find good elimination orderings. The first uses an important graph-theoretic property of induced graphs, and the second uses heuristic ideas.

9.4.3.1 Chordal Graphs

chordal graph

Recall from definition 2.24 that an undirected graph is *chordal* if it contains no cycle of length greater than three that has no "shortcut," that is, every minimal loop in the graph is of length three. As we now show, somewhat surprisingly, the class of induced graphs is equivalent to the class of chordal graphs. We then show that this property can be used to provide one heuristic for constructing an elimination ordering.

Theorem 9.8 *Every induced graph is chordal.*

PROOF Assume by contradiction that we have such a cycle $X_1 — X_2 — \ldots — X_k — X_1$ for $k > 3$, and assume without loss of generality that X_1 is the first variable to be eliminated. As in the proof of theorem 9.6, no edge incident on X_1 is added after X_1 is eliminated; hence, both edges $X_1 — X_2$ and $X_1 — X_k$ must exist at this point. Therefore, the edge $X_2 — X_k$ will be added at the same time, contradicting our assumption. ■

Indeed, we can verify that the graph of figure 9.11a is chordal. For example, the loop $H \rightarrow G \rightarrow L \rightarrow J \rightarrow H$ is cut by the chord $G \rightarrow J$.

The converse of this theorem states that any chordal graph $\mathcal{H}$ is an induced graph for some ordering. One way of showing that is to show that there is an elimination ordering for $\mathcal{H}$ for which $\mathcal{H}$ itself is the induced graph.

Theorem 9.9 *Any chordal graph $\mathcal{H}$ admits an elimination ordering that does not introduce any fill edges into the graph.*

PROOF We prove this result by induction on the number of nodes in the tree. Let $\mathcal{H}$ be a chordal graph with n nodes. As we showed in theorem 4.12, there is a clique tree $\mathcal{T}$ for $\mathcal{H}$. Let $\boldsymbol{C}_k$ be a clique in the tree that is a leaf, that is, it has only a single other clique as a neighbor. Let X_i be some variable that is in $\boldsymbol{C}_k$ but not in its neighbor. Let $\mathcal{H}'$ be the graph obtained by eliminating X_i. Because X_i belongs only to the clique $\boldsymbol{C}_k$, its neighbors are precisely $\boldsymbol{C}_k - \{X_i\}$. Because all of them are also in $\boldsymbol{C}_k$, they are connected to each other. Hence, eliminating X_i introduces no fill edges. Because $\mathcal{H}'$ is also chordal, we can now apply the inductive hypothesis, proving the result. ■

Algorithm 9.3 Maximum cardinality search for constructing an elimination ordering

```
    Procedure Max-Cardinality (
      H    // An undirected graph over X
    )
1     Initialize all nodes in X as unmarked
2     for k = |X| ... 1
3       X ← unmarked variable in X with largest number of marked neighbors
4       π(X) ← k
5       Mark X
6     return π
```

Example 9.2

We can illustrate this construction on the graph of figure 9.11a. The maximal cliques in the induced graph are shown in b, and a clique tree for this graph is shown in c. One can easily verify that each sepset separates the two sides of the tree; for example, the sepset $\{G, S\}$ separates C, I, D (on the left) from L, J, H (on the right). The elimination ordering C, D, I, H, G, S, L, J, an extension of the elimination in table 9.1 that generated this induced graph, is one ordering that might arise from the construction of theorem 9.9. For example, it first eliminates C, D, which are both in a leaf clique; it then eliminates I, which is in a clique that is now a leaf, following the elimination of C, D. Indeed, it is not hard to see that this ordering introduces no fill edges. By contrast, the ordering in table 9.2 is not consistent with this construction, since it begins by eliminating the variables G, I, S, none of which are in a leaf clique. Indeed, this elimination ordering introduces additional fill edges, for example, the edge $H \rightarrow D$. ■

maximum cardinality

An alternative method for constructing an elimination ordering that introduces no fill edges in a chordal graph is the Max-Cardinality algorithm, shown in algorithm 9.3. This method does not use the clique tree as its starting point, but rather operates directly on the graph. When applied to a chordal graph, it constructs an elimination ordering that eliminates cliques one at a time, starting from the leaves of the clique tree; and it does so without ever considering the clique tree structure explicitly.

Example 9.3

Consider applying Max-Cardinality *to the chordal graph of figure 9.11. Assume that the first node selected is S. The second node selected must be one of S's neighbors, say J. The node that has the largest number of marked neighbors are now G and L, which are chosen subsequently. Now, the unmarked nodes that have the largest number of marked neighbors (two) are H and I. Assume we select I. Then the next nodes selected are D and H, in any order. The last node to be selected is C. One possible resulting ordering in which nodes are marked is thus S, J, G, L, I, H, D, C. Importantly, the actual elimination ordering proceeds in reverse. Thus, we first eliminate C, D, then H, and so on. We can now see that this ordering always eliminates a variable from a clique that is a leaf clique at the time. For example, we first eliminate C, D from a leaf clique, then H, then G from the clique $\{G, I, D\}$, which is now (following the elimination of C, D) a leaf.* ■

As in this example, Max-Cardinality always produces an elimination ordering that is consistent with the construction of theorem 9.9. As a consequence, it follows that Max-Cardinality, when applied to a chordal graph, introduces no fill edges.

Theorem 9.10 *Let $\mathcal{H}$ be a chordal graph. Let π be the ranking obtained by running* Max-Cardinality *on $\mathcal{H}$. Then* Sum-Product-VE *(algorithm 9.1), eliminating variables in order of* increasing π*, does not introduce any fill edges.*

The proof is left as an exercise (exercise 9.8).

The maximum cardinality search algorithm can also be used to construct an elimination ordering for a nonchordal graph. However, it turns out that the orderings produced by this method are generally not as good as those produced by various other algorithms, such as those described in what follows.

To summarize, we have shown that, if we construct a chordal graph that contains the graph $\mathcal{H}_\Phi$ corresponding to our set of factors Φ, we can use it as the basis for inference using Φ. The process of turning a graph $\mathcal{H}$ into a chordal graph is also called *triangulation*, since it ensures that the largest unbroken cycle in the graph is a triangle. Thus, we can reformulate our goal of finding an elimination ordering as that of triangulating a graph $\mathcal{H}$ so that the largest clique in the resulting graph is as small as possible. Of course, this insight only reformulates the problem: Inevitably, the problem of finding such a minimal triangulation is also $\mathcal{NP}$-hard. Nevertheless, there are several graph-theoretic algorithms that address this precise problem and offer different levels of performance guarantee; we discuss this task further in section 10.4.2.

triangulation

Box 9.B — Concept: Polytrees. *One particularly simple class of chordal graphs is the class of Bayesian networks whose graph $\mathcal{G}$ is a* polytree*. Recall from definition 2.22 that a polytree is a graph where there is at most one trail between every pair of nodes.*

polytree

Polytrees received a lot of attention in the early days of Bayesian networks, because the first widely known inference algorithm for any type of Bayesian network was Pearl's message passing algorithm for polytrees. This algorithm, a special case of the message passing algorithms described in subsequent chapters of this book, is particularly compelling in the case of polytree networks, since it consists of nodes passing messages directly to other nodes along edges in the graph. Moreover, the cost of this computation is linear in the size of the network (where the size of the network is measured as the total sizes of the CPDs in the network, not the number of nodes; see exercise 9.9). From the perspective of the results presented in this section, this simplicity is not surprising: In a polytree, any maximal clique is a family of some variable in the network, and the clique tree structure roughly follows the network topology. (We simply throw out families that do not correspond to a maximal clique, because they are subsumed by another clique.)

Somewhat ironically, the compelling nature of the polytree algorithm gave rise to a long-standing misconception that there was a sharp tractability boundary between polytrees and other networks, in that inference was tractable only in polytrees and NP-hard in other networks. As we discuss in this chapter, this is not the case; rather, there is a continuum of complexity defined by the size of the largest clique in the induced graph.

9.4.3.2 Minimum Fill/Size/Weight Search

An alternative approach for finding elimination orderings is based on a very straightforward intuition. Our goal is to construct an ordering that induces a "small" graph. While we cannot

Algorithm 9.4 Greedy search for constructing an elimination ordering

```
Procedure Greedy-Ordering (
    H    // An undirected graph over X
    s    // An evaluation metric
)
1   Initialize all nodes in X as unmarked
2   for k = 1 ... |X|
3     Select an unmarked variable X ∈ X that minimizes s(H, X)
4     π(X) ← k
5     Introduce edges in H between all neighbors of X
6     Mark X
7   return π
```

find an ordering that achieves the global minimum, we can eliminate variables one at a time in a greedy way, so that each step tends to lead to a small blowup in size.

The general algorithm is shown in algorithm 9.4. At each point, the algorithm evaluates each of the remaining variables in the network based on its heuristic cost function. Some common cost criteria that have been used for evaluating variables are:

- **Min-neighbors:** The cost of a vertex is the number of neighbors it has in the current graph.
- **Min-weight:**The cost of a vertex is the product of *weights* — domain cardinality — of its neighbors.
- **Min-fill:** - The cost of a vertex is the number of edges that need to be added to the graph due to its elimination.
- **Weighted-min-fill:** The cost of a vertex is the sum of weights of the edges that need to be added to the graph due to its elimination, where a weight of an edge is the product of weights of its constituent vertices.

Intuitively, min-neighbors and min-weight count the size or weight of the largest clique in $\mathcal{H}$ after eliminating X. Min-fill and weighted-min-fill count the number or weight of edges that would be introduced into $\mathcal{H}$ by eliminating X. It can be shown (exercise 9.10) that none of these criteria is universally better than the others.

This type of greedy search can be done either deterministically (as shown in algorithm 9.4), or stochastically. In the stochastic variant, at each step we select some number of low-scoring vertices, and then choose among them using their score (where lower-scoring vertices are selected with higher probability). In the stochastic variants, we run multiple iterations of the algorithm, and then select the ordering that leads to the most efficient elimination — the one where the sum of the sizes of the factors produced is smallest.

Empirical results show that these heuristic algorithms perform surprisingly well in practice. Generally, Min-Fill and Weighted-Min-Fill tend to work better on more problems. Not surprisingly, Weighted-Min-Fill usually has the most significant gains when there is some significant variability in the sizes of the domains of the variables in the network. Box 9.C presents a case study comparing these algorithms on a suite of standard benchmark networks.

Box 9.C — Case Study: Variable Elimination Orderings. *Fishelson and Geiger (2003) performed a comprehensive case study of different heuristics for computing an elimination ordering, testing them on eight standard Bayesian network benchmarks, ranging from 24 nodes to more than 1,000. For each network, they compared both to the best elimination ordering known previously, obtained by an expensive process of simulated annealing search, and to the network obtained by a state-of-the-art Bayesian network package. They compared to stochastic versions of the four heuristics described in the text, running each of them for 1 minute or 10 minutes, and selecting the best network obtained in the different random runs. Maximum cardinality search was not used, since it is known to perform quite poorly in practice.*

The results, shown in figure 9.C.1, suggest several conclusions. First, we see that running the stochastic algorithms for longer improves the quality of the answer obtained, although usually not by a huge amount. We also see that different heuristics can result in orderings whose computational cost can vary in almost an order of magnitude. Overall, Min-Fill and Weighted-Min-Fill achieve the best performance, but they are not universally better. The best answer obtained by the greedy algorithms is generally very good; it is often significantly better than the answer obtained by a deterministic state-of-the-art scheme, and it is usually quite close to the best-known ordering, even when the latter is obtained using much more expensive techniques. Because the computational cost of the heuristic ordering-selection algorithms is usually negligible relative to the running time of the inference itself, we conclude that for large networks it is worthwhile to run several heuristic algorithms in order to find the best ordering obtained by any of them.

9.5 Conditioning ⋆

conditioning

An alternative approach to inference is based on the idea of *conditioning*. The conditioning algorithm is based on the fact (illustrated in section 9.3.2), that observing the value of certain variables can simplify the variable elimination process. When a variable is not observed, we can use a case analysis to enumerate its possible values, perform the simplified VE computation, and then aggregate the results for the different values. As we will discuss, **in terms of number of operations, the conditioning algorithm offers no benefit over the variable elimination algorithm. However, it offers a continuum of time-space trade-offs, which can be extremely important in cases where the factors created by variable elimination are too big to fit in main memory.**

9.5.1 The Conditioning Algorithm

The conditioning algorithm is easiest to explain in the context of a Markov network. Let Φ be a set of factors over $\boldsymbol{X}$ and P_Φ be the associated distribution. We assume that any observations were already assimilated into Φ, so that our goal is to compute $P_\Phi(\boldsymbol{Y})$ for some set of query variables $\boldsymbol{Y}$. For example, if we want to do inference in the Student network given the evidence $G = g$, we would reduce the factors reduced to this context, giving rise to the network structure shown in figure 4.6b.

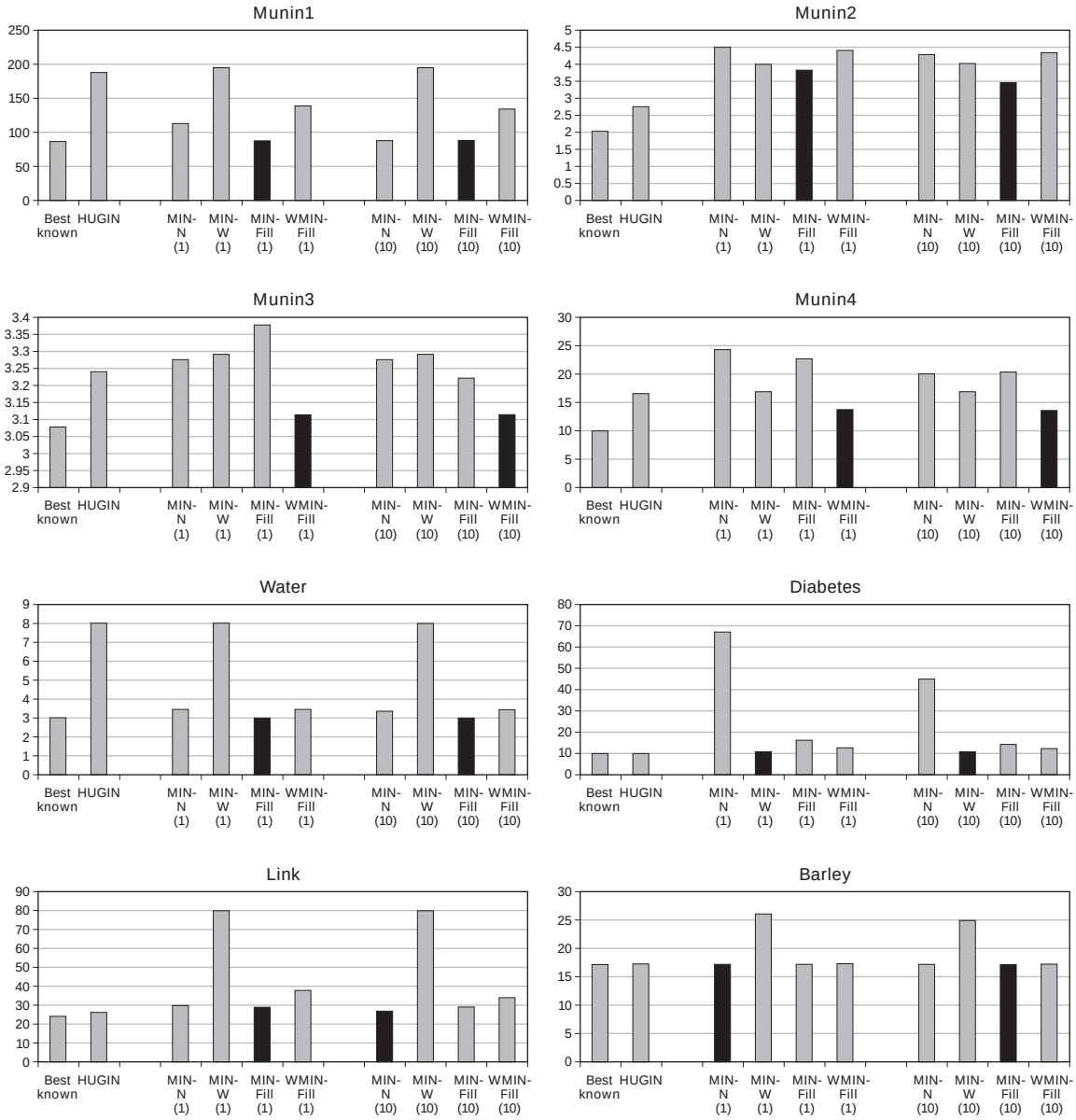


Figure 9.C.1 — Comparison of algorithms for selecting variable elimination ordering. Computational cost of variable elimination inference in a range of benchmark networks, obtained by various algorithms for selecting an elimination ordering. The cost is measured as the size of the factors generated during the process of variable elimination. For each network, we see the cost of the best-known ordering, the ordering obtained by HUGIN (a state-of-the-art Bayesian network package), and the ordering obtained by stochastic greedy search using four different search heuristics — Min-Neighbors, Min-Weight, Min-Fill, and Weighted-Min-Fill — run for 1 minute and for 10 minutes.

Algorithm 9.5 Conditioning algorithm

Procedure Sum-Product-Conditioning (
  Φ, // Set of factors, possibly reduced by evidence
  $\boldsymbol{Y}$, // Set of query variables
  $\boldsymbol{U}$ // Set of variables on which to condition
)
1 **for** each $\boldsymbol{u} \in Val(\boldsymbol{U})$
2 $\quad \Phi_{\boldsymbol{u}} \leftarrow \{\phi[\boldsymbol{U} = \boldsymbol{u}] \ : \ \phi \in \Phi\}$
3 $\quad$ Construct $\mathcal{H}_{\Phi_{\boldsymbol{u}}}$
4 $\quad (\alpha_{\boldsymbol{u}}, \phi_{\boldsymbol{u}}(\boldsymbol{Y})) \leftarrow$ Cond-Prob-VE$(\mathcal{H}_{\Phi_{\boldsymbol{u}}}, \boldsymbol{Y}, \emptyset)$
5 $\phi^*(\boldsymbol{Y}) \leftarrow \frac{\sum_{\boldsymbol{u}} \phi_{\boldsymbol{u}}(\boldsymbol{Y})}{\sum_{\boldsymbol{u}} \alpha_{\boldsymbol{u}}}$
6 Return $\phi^*(\boldsymbol{Y})$

The conditioning algorithm is based on the following simple derivation. Let $\boldsymbol{U} \subseteq \boldsymbol{X}$ be any set of variables. Then we have that:

$$\tilde{P}_\Phi(\boldsymbol{Y}) = \sum_{\boldsymbol{u} \in Val(\boldsymbol{U})} \tilde{P}_\Phi(\boldsymbol{Y}, \boldsymbol{u}). \tag{9.11}$$

The key observation is that each term $\tilde{P}_\Phi(\boldsymbol{Y}, \boldsymbol{u})$ can be computed by marginalizing out the variables in $\boldsymbol{X} - \boldsymbol{U} - \boldsymbol{Y}$ *in the unnormalized measure* $\tilde{P}_\Phi[\boldsymbol{u}]$ *obtained by reducing* $\tilde{P}_\Phi$ *to the context* $\boldsymbol{u}$. As we have already discussed, the reduced measure is simply the measure defined by reducing each of the factors to the context $\boldsymbol{u}$. The reduction process generally produces a simpler structure, with a reduced inference cost.

We can use this formula to compute $P_\Phi(\boldsymbol{Y})$ as follows: We construct a network $\mathcal{H}_\Phi[\boldsymbol{u}]$ for each assignment $\boldsymbol{u}$; these networks have identical structures, but different parameters. We run sum-product inference in each of them, to obtain a factor over the desired query set $\boldsymbol{Y}$. We then simply add up these factors to obtain $\tilde{P}_\Phi(\boldsymbol{Y})$. We can also derive $P_\Phi(\boldsymbol{Y})$ by renormalizing this factor to obtain a distribution. As usual, the normalizing constant is the partition function for P_Φ. However, applying equation (9.11) to the case of $\boldsymbol{Y} = \emptyset$, we conclude that

$$Z_\Phi = \sum_{\boldsymbol{u}} Z_{\Phi[\boldsymbol{u}]}.$$

Thus, we can derive the overall partition function from the partition functions for the different subnetworks $\mathcal{H}_{\Phi[\boldsymbol{u}]}$. The final algorithm is shown in algorithm 9.5. (We note that Cond-Prob-VE was called without evidence, since we assumed for simplicity that our factors Φ have already been reduced with the evidence.)

Example 9.4

Assume that we want to compute $P(J)$ in the Student *network with evidence $G = g^1$, so that our initial graph would be the one shown in figure 4.6b. We can now perform inference by enumerating all of the assignments s to the variable S. For each such assignment, we run inference on a graph structured as in figure 4.6c, with the factors reduced to the assignment g^1, s. In each such network we compute a factor over J, and add them all up. Note that the reduced network contains two disconnected components, and so we might be tempted to run inference only on the component that contains J. However, that procedure would not produce a correct answer: The value we get by summing out the variables in the second component multiplies our final factor. Although this is a constant multiple for each value of s, these values are generally different for the different values of S. Because the factors are added before the final renormalization, this constant influences the weight of one factor in the summation relative to the other. Thus, if we ignore this constant component, the answers we get from the s^1 computation and the s^0 computation would be weighted incorrectly.* ■

Historically, owing to the initial popularity of the polytree algorithm, the conditioning approach was mostly used in the case where the transformed network is a polytree. In this case, the algorithm is called *cutset conditioning*.

cutset conditioning

9.5.2 Conditioning and Variable Elimination

At first glance, it might appear as if this process saves us considerable computational cost over the variable elimination algorithm. After all, we have reduced the computation to one that performs variable elimination in a much simpler network. The cost arises, of course, from the fact that, when we condition on $\boldsymbol{U}$, we need to perform variable elimination on the conditioned network multiple times, once for each assignment $\boldsymbol{u} \in Val(\boldsymbol{U})$. The cost of this computation is $O(|Val(\boldsymbol{U})|)$, which is exponential in the number of variables in $\boldsymbol{U}$. Thus, we have not avoided the exponential blowup associated with the probabilistic inference process. In this section, we provide a formal complexity analysis of the conditioning algorithm, and compare it to the complexity of elimination. This analysis also reveals various interesting improvements to the basic conditioning algorithm, which can dramatically improve its performance in certain cases.

To understand the operation of the conditioning algorithm, we return to the basic description of the probabilistic inference task. Consider our query J in the Extended Student network. We know that:

$$p(J) = \sum_C \sum_D \sum_I \sum_S \sum_G \sum_L \sum_H P(C, D, I, S, G, L, H, J).$$

Reordering this expression slightly, we have that:

$$p(J) = \sum_g \left[\sum_C \sum_D \sum_I \sum_S \sum_L \sum_H P(C, D, I, S, g, L, H, J) \right].$$

The expression inside the parentheses is precisely the result of computing the probability of J in the network $\mathcal{H}_{\Phi_{G=g}}$, where Φ is the set of CPD factors in $\mathcal{B}$.

In other words, the conditioning algorithm is simply executing parts of the basic summation defining the inference task by case analysis, enumerating the possible values of the conditioning

Step	Variable eliminated	Factors used	Variables involved	New factor
1	C	$\phi_C^+(C,G)$, $\phi_D^+(D,C,G)$	C,D,G	$\tau_1(D,G)$
2	D	$\phi_G^+(G,I,D)$, $\tau_1(D,G)$	G,I,D	$\tau_2(G,I)$
3	I	$\phi_I^+(I,G)$, $\phi_S^+(S,I,G)$, $\tau_2(G,I)$	G,S,I	$\tau_3(G,S)$
4	H	$\phi_H^+(H,G,J)$	H,G,J	$\tau_4(G,J)$
5	S	$\tau_3(G,S)$, $\phi_J^+(J,L,S,G)$	J,L,S,G	$\tau_5(J,L,G)$
6	L	$\tau_5(J,L,G)$, $\phi_L^+(L,G)$	J,L	$\tau_6(J)$
7	—	$\tau_6(J)$, $\tau_4(G,J)$	G,J	$\tau_7(G,J)$

Table 9.4 Example of relationship between variable elimination and conditioning. A run of variable elimination for the query $P(J)$ corresponding to conditioning on G.

variables. By contrast, variable elimination performs the same summation from the inside out, using dynamic programming to reuse computation.

Indeed, if we simply did conditioning on all of the variables, the result would be an explicit summation of the entire joint distribution. In conditioning, however, we perform the conditioning step only on some of the variables, and use standard variable elimination — dynamic programming — to perform the rest of the summation, avoiding exponential blowup (at least over that part).

In general, it follows that both algorithms are performing the same set of basic operations (sums and products). However, where the variable elimination algorithm uses the caching of dynamic programming to save redundant computation throughout the summation, conditioning uses a full enumeration of cases for some of the variables, and dynamic programming only at the end.

From this argument, it follows that conditioning always performs no fewer steps than variable elimination. To understand why, consider the network of example 9.4 and assume that we are trying to compute $P(J)$. The conditioned network $\mathcal{H}_{\Phi_{G=g}}$ has a set of factors most of which are identical to those in the original network. The exceptions are the reduced factors: $\phi_L[G=g](L)$ and $\phi_H[G=g](H,J)$. For each of the three values g of G, we are performing variable elimination over these factors, eliminating all variables except for G and J.

We can imagine "lumping" these three computations into one, by augmenting the scope of each factor with the variable G. More precisely, we define a set of augmented factors ϕ^+ as follows: The scope of the factor ϕ_G already contains G, so $\phi_G^+(G,D,I) = \phi_G(G,D,I)$. For the factor ϕ_L^+, we simply combine the three factors $\phi_{L,g}(L)$, so that $\phi_L^+(L,g) = \phi_L[G=g](L)$ for all g. Not surprisingly, the resulting factor $\phi_L^+(L,G)$ is simply our original CPD factor $\phi_L(L,G)$. We define ϕ_H^+ in the same way. The remaining factors are unrelated to G. For each other variable X over scope $\boldsymbol{Y}$, we simply define $\phi_X^+(\boldsymbol{Y},G) = \phi_X(\boldsymbol{Y})$; that is, the value of the factor does not depend on the value of G.

We can easily verify that, if we run variable elimination over the set of factors $\mathcal{F}_X^+$ for $X \in \{C,D,I,G,S,L,J,H\}$, eliminating all variables except for J and G, we are performing precisely the same computation as the three iterations of variable elimination for the three different conditioned networks $\mathcal{H}_{\Phi_{G=g}}$: Factor entries involving different values g of G never in-

Step	Variable eliminated	Factors used	Variables involved	New factor
1	C	$\phi_C(C)$, $\phi_D(D, C)$	C, D	$\tau_1(D)$
2	D	$\phi_G(G, I, D)$, $\tau_1(D)$	G, I, D	$\tau_2(G, I)$
3	I	$\phi_I(I)$, $\phi_S(S, I)$, $\tau_2(G, I)$	G, S, I	$\tau_3(G, S)$
4	H	$\phi_H(H, G, J)$	H, G, J	$\tau_4(G, J)$
5	S	$\tau_3(G, S)$, $\phi_J(J, L, S)$	J, L, S, G	$\tau_5(J, L, G)$
6	L	$\tau_5(J, L, G)$, $\phi_L(L, G)$	J, L	$\tau_6(J)$
7	G	$\tau_6(J)$, $\tau_4(G, J)$	G, J	$\tau_7(J)$

Table 9.5 A run of variable elimination for the query $P(J)$ with G eliminated last

teract, and the computation performed for the entries where $G = g$ is precisely the computation performed in the network $\mathcal{H}_{\Phi_{G=g}}$.

Specifically, assume we are using the ordering C, D, I, H, S, L to perform the elimination within each conditioned network $\mathcal{H}_{\Phi_{G=g}}$. The steps of the computation are shown in table 9.4. Step (7) corresponds to the product of all of the remaining factors, which is the last step in variable elimination. The final step in the conditioning algorithm, where we add together the results of the three computations, is precisely the same as eliminating G from the resulting factor $\tau_7(G, J)$.

It is instructive to compare this execution to the one obtained by running variable elimination on the original set of factors, with the elimination ordering C, D, I, H, S, L, G; that is, we follow the ordering used within the conditioned networks for the variables other than G, J, and then eliminate G at the very end. In this process, shown in table 9.5, some of the factors involve G, but others do not. In particular, step (1) in the elimination algorithm involves only C, D, whereas in the conditioning algorithm, we are performing precisely the same computation over C, D three times: once for each value g of G.

In general, we can show:

Theorem 9.11

Let Φ be a set of factors, and $\boldsymbol{Y}$ be a query. Let $\boldsymbol{U}$ be a set of conditioning variables, and $\boldsymbol{Z} = \mathcal{X} - \boldsymbol{Y} - \boldsymbol{U}$. Let $\prec$ be the elimination ordering over $\boldsymbol{Z}$ used by the variable elimination algorithm over the network $\mathcal{H}_{\Phi_u}$ in the conditioning algorithm. Let $\prec^+$ be an ordering that is consistent with $\prec$ over the variables in $\boldsymbol{Z}$, and where, for each variable $U \in \boldsymbol{U}$, we have that $\boldsymbol{Z} \prec^+ U$. Then the number of operations performed by the conditioning is no less than the number of operations performed by variable elimination with the ordering $\prec^+$.

We omit the proof of this theorem, which follows precisely the lines of our example.

Thus, conditioning always requires no fewer computations than variable elimination with some particular ordering (which may or may not be a good one). In our example, the wasted computation from conditioning is negligible. In other cases, however, as we will discuss, we can end up with a large amount of redundant computation. In fact, in some cases, conditioning can be significantly worse:

Example 9.5

Consider the network shown in figure 9.12a, and assume we choose to condition on A_k in order

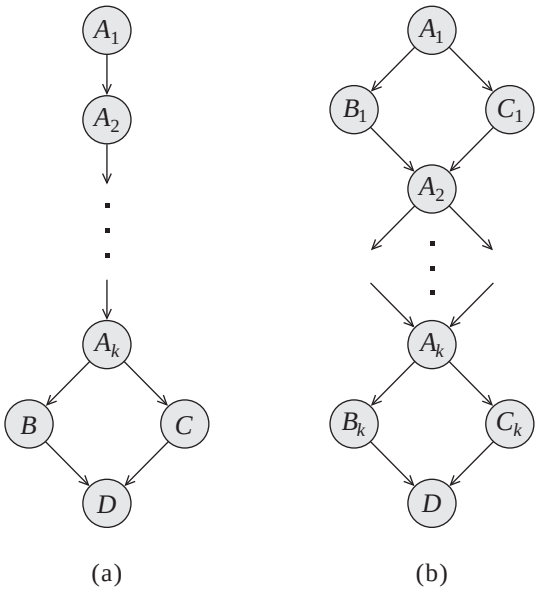


Figure 9.12 Networks where conditioning performs unnecessary computation

to cut the single loop in the network. In this case, we would perform the entire elimination of the chain $A_1 \rightarrow \ldots \rightarrow A_{k-1}$ multiple times — once for every value of A_k. ■

Example 9.6

Consider the network shown in figure 9.12b and assume that we wish to use cutset conditioning, where we cut every loop in the network. The most efficient way of doing so is to condition on every other A_i variable, for example, $A_2, A_4, \ldots, A_k$ (assuming for simplicity that k is even). The cost of the conditioning algorithm in this case is exponential in k, whereas the induced width of the network is 2, and the cost of variable elimination is linear in k. ■

Given this discussion, one might wonder why anyone bothers with the conditioning algorithm. There are two main reasons. First, variable elimination gains its computational savings from caching factors computed as intermediate results. In complex networks, these factors can grow very large. In cases where memory is scarce, it might not be possible to keep these factors in memory, and the variable elimination computation becomes infeasible (or very costly due to constant thrashing to disk). On the other hand, conditioning does not require significant amounts of memory: We run inference separately for each assignment $\boldsymbol{u}$ to $\boldsymbol{U}$ and simply accumulate the results. Overall, the computation requires space that is linear only in the size of the network. Thus, we can view the trade-off of conditioning versus variable elimination as a time-space trade-off. Conditioning saves space by not storing intermediate results in memory, but then it may cost additional time by having to repeat the computation to generate them.

The second reason for using conditioning is that it forms the basis for a useful approximate inference algorithm. In particular, in certain cases, we can get a reasonable approximate solution

by enumerating only some of the possible assignment $\boldsymbol{u} \in Val(\boldsymbol{U})$. We return to this approach in section 12.5

9.5.3 Graph-Theoretic Analysis

As in the case of variable elimination, it helps to reformulate the complexity analysis of the conditioning algorithm in graph-theoretic terms. Assume that we choose to condition on a set $\boldsymbol{U}$, and perform variable elimination on the remaining variables. We can view each of these steps in terms of its effect on the graph structure.

Let us begin with the step of conditioning the network on some variable U. Once again, it is easiest to view this process in terms of its effect on an undirected graph. As we discussed, this step effectively introduces U into every factor parameterizing the current graph. In graph-theoretic terms, we have introduced U into every clique in the graph, or, more simply, introduced an edge between U and every other node currently in the graph.

When we finish the conditioning process, we perform elimination on the remaining variables. We have already analyzed the effect on the graph of eliminating a variable X: When we eliminate X, we add edges between all of the current neighbors of X in the graph. We then remove X from the graph.

We can now define an induced graph for the conditioning algorithm. Unlike the graph for variable elimination, this graph has two types of fill edges: those induced by conditioning steps, and those induced by the elimination steps for the remaining variables.

Definition 9.7 conditioning induced graph

Let Φ be a set of factors over $\mathcal{X} = \{X_1, \ldots, X_n\}$, $\boldsymbol{U} \subset \mathcal{X}$ be a set of conditioning variables, and $\prec$ be an elimination ordering for some subset $\boldsymbol{X} \subseteq \mathcal{X} - \boldsymbol{U}$. The induced graph $\mathcal{I}_{\Phi,\prec,\boldsymbol{U}}$ *is an undirected graph over $\mathcal{X}$ with the following edges:*

- *a* conditioning edge *between every variable $U \in \boldsymbol{U}$ and every other variable $X \in \mathcal{X}$;*
- *a* factor edge *between every pair of variables $X_i, X_j \in \boldsymbol{X}$ that both appear in some intermediate factor ψ generated by the VE algorithm using $\prec$ as an elimination ordering.* ■

Example 9.7

Consider the Student *example of figure 9.8, where our query is $P(J)$. Assume that (for some reason) we condition on the variable L and perform elimination on the remaining variables using the ordering C, D, I, H, G, S. The graph induced by this conditioning set and this elimination ordering is shown in figure 9.13, with the conditioning edges shown as dashed lines and the factor edges shown, as usual, by complete lines. The step of conditioning on L causes the introduction of the edges between L and all the other variables. The set of factors we have after the conditioning step immediately leads to the introduction of all the factor edges except for the edge G—S; this latter edge results from the elimination of I.* ■

We can now use this graph to analyze the complexity of the conditioning algorithm.

Theorem 9.12

Consider an application of the conditioning algorithm to a set of factors Φ, where $\boldsymbol{U} \subset \mathcal{X}$ is the set of conditioning variables, and $\prec$ is the elimination ordering used for the eliminated variables $\boldsymbol{X} \subseteq \mathcal{X} - \boldsymbol{U}$. Then the running time of the algorithm is $O(n \cdot v^m)$, where v is a bound on the domain size of any variable, and m is the size of the largest clique in the graph, using both conditioning and factor edges.

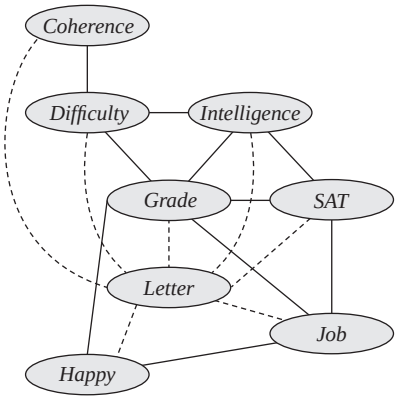


Figure 9.13 Induced graph for the Student **example using both conditioning and elimination:** we condition on L and eliminate the remaining variables using the ordering C, D, I, H, G, S.

The proof is left as an exercise (exercise 9.12).

This theorem provides another perspective on the trade-off between conditioning and elimination in terms of their time complexity. Consider, as we did earlier, an algorithm that simply defers the elimination of the conditioning variables $\boldsymbol{U}$ until the end. Consider the effect on the graph of the earlier steps of the elimination algorithm (those preceding the elimination of $\boldsymbol{U}$). As variables are eliminated, certain edges might be added between the variables in $\boldsymbol{U}$ and other variables (in particular, we add an edge between X and $U \in \boldsymbol{U}$ whenever they are both neighbors of some eliminated variable Y). However, conditioning adds edges between the variables $\boldsymbol{U}$ and *all* other variables X. Thus, conditioning always results in a graph that contains at least as many edges as the induced graph from elimination using this ordering.

However, we can also use the same graph to precisely estimate the time-space trade-off provided by the conditioning algorithm.

Theorem 9.13

Consider an application of the conditioning algorithm to a set of factors Φ, where $\boldsymbol{U} \subset \mathcal{X}$ is the set of conditioning variables, and $\prec$ is the elimination ordering used for the eliminated variables $\boldsymbol{X} \subseteq \mathcal{X} - \boldsymbol{U}$. The space complexity of the algorithm is $O(n \cdot v^{m_f})$, where v is a bound on the domain size of any variable, and m_f is the size of the largest clique in the graph using only factor edges.

The proof is left as an exercise (exercise 9.13).

By comparison, the asymptotic space complexity of variable elimination is the same as its time complexity: exponential in the size of the largest clique containing both types of edges. Thus, we see precisely that conditioning allows us to perform the computation using less space, at the cost (usually) of additional running time.

9.5.4 Improved Conditioning

As we discussed, in terms of the total operations performed, conditioning cannot be better than variable elimination. As we now show, conditioning, naively applied, can be significantly worse.

However, the insights gained from these examples can be used to improve the conditioning algorithm, reducing its cost significantly in many cases.

9.5.4.1 Alternating Conditioning and Elimination

As we discussed, the main problem associated with conditioning is the fact that all computations are repeated for all values of the conditioning variables, even in cases where the different computations are, in fact, identical. This phenomenon arose in the network of example 9.5.

It seems clear, in this example, that we would prefer to eliminate the chain $A_1 \rightarrow \ldots \rightarrow A_{k-1}$ once and for all, before conditioning on A_k. Having eliminated the chain, we would then end up with a much simpler network, involving factors only over A_k, B, C, and D, to which we can then apply conditioning.

The perspective described in section 9.5.3 provides the foundation for implementing this idea. As we discussed, variable elimination works from the inside out, summing out variables in the innermost summation first and caching the results. On the other hand, conditioning works from the outside in, performing the entire internal summation (using elimination) for each value of the conditioning variables, and only then summing the results. However, there is nothing that forces us to split our computation on the outermost summations before considering the inner ones. Specifically, we can eliminate one or more variables on the inside of the summation before conditioning on any variable on the outside.

Example 9.8 *Consider again the network of figure 9.12a, and assume that our goal is to compute $P(D)$. We might formulate the expression as:*

$$\sum_{A_k} \sum_{B} \sum_{C} \sum_{A_1} \cdots \sum_{A_{k-1}} P(A_1, \ldots, A_k, B, C, D).$$

We can first perform the internal summations on $A_{k-1}, \ldots, A_1$, resulting in a set of factors over the scope A_k, B, C, D. We can now condition this network (that is, the Markov network induced by the resulting set of factors) on A_k, resulting in a set of simplified networks over B, C, D (one for each value of A_k). In each such network, we use variable elimination on B and C to compute a factor over D, and aggregate the factors from the different networks, as in standard conditioning. ■

In this example, we first perform some elimination, then condition, and then elimination on the remaining network. Clearly, we can generalize this idea to define an algorithm that alternates the operations of elimination and conditioning arbitrarily. (See exercise 9.14.)

9.5.4.2 Network Decomposition

A second class of examples where we can significantly improve the performance of conditioning arises in networks where conditioning on some subset of variables splits the graph into independent pieces.

Example 9.9 *Consider the network of example 9.6, and assume that $k = 16$, and that we begin by conditioning on A_2. After this step, the network is decomposed into two independent pieces. The standard conditioning algorithm would continue by conditioning further, say on A_3. However, there is really no need to condition the top part of the network — the one associated with the variables*

A_1, B_1, C_1 on the variable A_3: none of the factors mention A_3, and we would be repeating exactly the same computation for each of its values. ■

Clearly, having partitioned the network into two completely independent pieces, we can now perform the computation on each of them separately, and then combine the results. In particular, the conditioning variables used on one part would not be used at all to condition the other. More precisely, we can define an algorithm that checks, after each conditioning step, whether the resulting set of factors has been disconnected or not. If it has, it simply partitions them into two or more disjoint sets and calls the algorithm recursively on each subset.

9.6 Inference with Structured CPDs ⋆

We have seen that BN inference exploits the network structure, in particular the conditional independence and the locality of influence. But when we discussed representation, we also allowed for the representation of finer-grained structure within the CPDs. It turns out that a carefully designed inference algorithm can also exploit certain types of local CPD structure. We focus on two types of structure where this issue has been particularly well studied — independence of causal influence, and asymmetric dependencies — using each of them to illustrate a different type of method for exploiting local structure in variable elimination. We defer the discussion of inference in networks involving continuous variables to chapter 14.

9.6.1 Independence of Causal Influence

The earliest and simplest instance of exploiting local structure was for CPDs that exhibit independence of causal influence, such as noisy-or.

9.6.1.1 Noisy-Or Decompositions

Consider a simple network consisting of a binary variable Y and its four binary parents X_1, X_2, X_3, X_4, where the CPD of Y is a noisy-or. Our goal is to compute the probability of Y. The operations required to execute this process, assuming we use an optimal ordering, is:

- 4 multiplications for $P(X_1) \cdot P(X_2)$
- 8 multiplications for $P(X_1, X_2) \cdot P(X_3)$
- 16 multiplications for $P(X_1, X_2, X_3) \cdot P(X_4)$
- 32 multiplications for $P(X_1, X_2, X_3, X_4) \cdot P(Y \mid X_1, X_2, X_3, X_4)$

The total is 60 multiplications, plus another 30 additions to sum out $X_1, \ldots, X_4$, in order to reduce the resulting factor $P(X_1, X_2, X_3, X_4, Y)$, of size 32, into the factor $P(Y)$ of size 2.

However, we can exploit the structure of the CPD to substantially reduce the amount of computation. As we discussed in section 5.4.1, a noisy-or variable can be decomposed into a deterministic OR of independent noise variables, resulting in the subnetwork shown in figure 9.14a. This transformation, by itself, is not very helpful. The factor $P(Y \mid Z_1, Z_2, Z_3, Z_4)$ is still of size 32 if we represent it as a full factor, so we achieve no gains.

The key idea is that the deterministic OR variable can be decomposed into various cascades of deterministic OR variables, each with a very small indegree. Figure 9.14b shows a simple

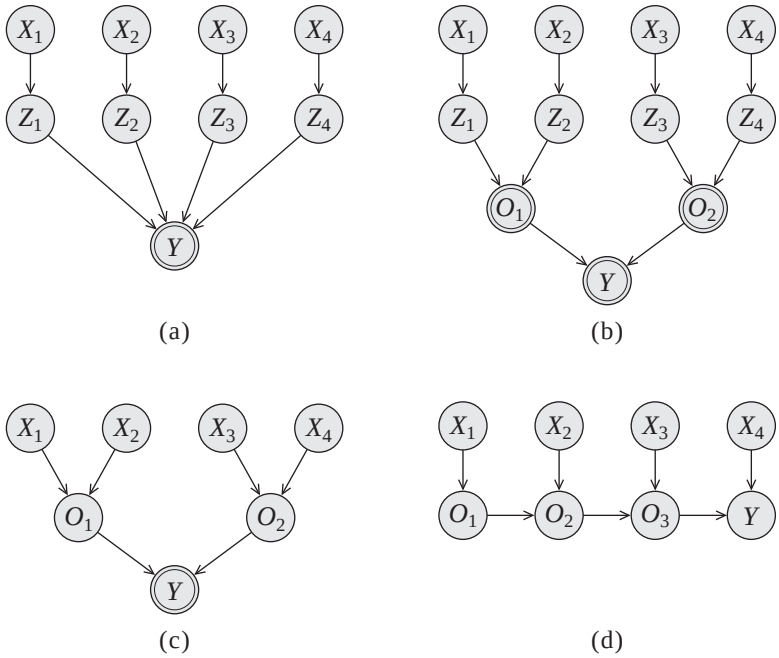


Figure 9.14 Different decompositions for a noisy-or CPD: (a) The standard decomposition of a noisy-or. (b) A tree decomposition of the deterministic-or. (c) A tree-based decomposition of the noisy-or. (d) A chain-based decomposition of the noisy-or.

decomposition of the deterministic OR as a tree. We can simplify this construction by eliminating the intermediate variables Z_i, integrating the "noise" for each X_i into the appropriate O_i. In particular, O_1 would be the noisy-or of X_1 and X_2, with the original noise parameters and a leak parameter of 0. The resulting construction is shown in figure 9.14c.

We can now revisit the inference task in this apparently more complex network. An optimal ordering for variable elimination is $X_1, X_2, X_3, X_4, O_1, O_2$. The cost of performing elimination of X_1, X_2 is:

- 8 multiplications for $\psi_1(X_1, X_2, O_1) = P(X_1) \cdot P(O_1 \mid X_1, X_2)$
- 4 additions to sum out X_1 in $\tau_1(X_2, O_1) = \sum_{X_1} \psi_1(X_1, X_2, O_1)$
- 4 multiplications for $\psi_2(X_2, O_1) = \tau_1(X_2, O_1) \cdot P(X_2)$
- 2 additions for $\tau_2(O_1) = \sum_{X_2} \psi_2(X_2, O_1)$

The cost for eliminating X_3, X_4 is identical, as is the cost for subsequently eliminating O_1, O_2. Thus, the total number of operations is $3 \cdot (8 + 4) = 36$ multiplications and $3 \cdot (4 + 2) = 18$ additions.

A different decomposition of the OR variable is as a simple cascade, where each Z_i is consecutively OR'ed with the previous intermediate result. This decomposition leads to the construction

of figure 9.14d. For this construction, an optimal elimination ordering is $X_1, O_1, X_2, O_2, X_3, O_3, X_4$. A simple analysis shows that it takes 4 multiplications and 2 additions to eliminate each of $X_1, \ldots, X_4$, and 8 multiplications and 4 additions to eliminate each of O_1, O_2, O_3. The total cost is $4 \cdot 4 + 3 \cdot 8 = 40$ multiplications and $4 \cdot 2 + 3 \cdot 4 = 20$ additions.

9.6.1.2 The General Decomposition

Clearly, the construction used in the preceding example is a general one that can be applied to more complex networks and other types of CPDs that have independence of causal influence. We take a variable whose CPD has independence of causal influence, and generate its decomposition into a set of independent noise models and a deterministic function, as in figure 5.13.

We then cascade the computation of the deterministic function into a set of smaller steps. Given our assumption about the symmetry and associativity of the deterministic function in the definition of symmetric ICI (definition 5.13), any decomposition of the deterministic function results in the same answer. Specifically, consider a variable Y with parents $X_1, \ldots, X_k$, whose CPD satisfies definition 5.13. We can decompose Y by introducing $k-1$ intermediate variables $O_1, \ldots, O_{k-1}$, such that:

- the variable Z, and each of the O_i's, has exactly two parents in $Z_1, \ldots, Z_k, O_1, \ldots, O_{i-1}$;
- the CPD of Z and of O_i is the deterministic $\diamond$ of its two parents;
- each Z_l and each O_i is a parent of at most one variable in $O_1, \ldots, O_{k-1}, Z$.

These conditions ensure that $Z = Z_1 \diamond Z_2 \diamond \ldots \diamond Z_k$, but that this function is computed gradually, where the node corresponding to each intermediate result has an indegree of 2.

We note that we can save some extraneous nodes, as in our example, by aggregating the noisy dependence of Z_i on X_i into the CPD where Z_i is used.

After executing this decomposition for every ICI variable in the network, we can simply apply variable elimination to the decomposed network with the smaller factors. As we saw, the complexity of the inference can go down substantially if we have smaller CPDs and thereby smaller factors.

We note that the sizes of the intermediate factors depend not only on the number of variables in their scope, but also on the domains of these variables. For the case of noisy-or variables (as well as noisy-max, noisy-and, and so on), the domain size of these variables is fixed and fairly small. However, in other cases, the domain might be quite large. In particular, in the case of generalized linear models, the domain of the intermediate variable Z generally grows linearly with the number of parents.

Example 9.10

Consider a variable Y with $\mathrm{Pa}_Y = \{X_1, \ldots, X_k\}$, where each X_i is binary. Assume that Y's CPD is a generalized linear model, whose parameters are $w_0 = 0$ and $w_i = w$ for all $i > 1$. Then the domain of the intermediate variable Z is $\{0, 1, \ldots, k\}$. In this case, the decomposition provides a trade-off: The size of the original CPD for $P(Y \mid X_1, \ldots, X_k)$ grows as 2^k; the size of the factors in the decomposed network grow roughly as k^3. In different situations, one approach might be better than the other. ■

Thus, the decomposition of symmetric ICI variables might not always be beneficial.

9.6.1.3 Global Structure

Our decomposition of the function f that defines the variable Z can be done in many ways, all of which are equivalent in terms of their final result. However, they are not equivalent from the perspective of computational cost. Even in our simple example, we saw that one decomposition can result in fewer operations than the other. The situation is significantly more complicated when we take into consideration other dependencies in the network.

Example 9.11

Consider the network of figure 9.14c, and assume that X_1 and X_2 have a joint parent A. In this case, we eliminate A first, and end up with a factor over X_1, X_2. Aside from the $4 + 8 = 12$ multiplications and 4 additions required to compute this factor $\tau_0(X_1, X_2)$, it now takes 8 multiplications to compute $\psi_1(X_1, X_2, O_1) = \tau_0(X_1, X_2) \cdot P(O_1 \mid X_1, X_2)$, and $4 + 2 = 6$ additions to sum out X_1 and X_2 in ψ_1. The rest of the computation remains unchanged. Thus, the total number of operations required to eliminate all of $X_1, \ldots, X_4$ (after the elimination of A) is $8 + 12 = 20$ multiplications and $6 + 6 = 12$ additions.

Conversely, assume that X_1 and X_3 have the joint parent A. In this case, it still requires 12 multiplications and 4 additions to compute a factor $\tau_0(X_1, X_3)$, but the remaining operations become significantly more complex. In particular, it takes:

- *8 multiplications for $\psi_1(X_1, X_2, X_3) = \tau_0(X_1, X_3) \cdot P(X_2)$*
- *16 multiplications for $\psi_2(X_1, X_2, X_3, O_1) = \psi_1(X_1, X_2, X_3) \cdot P(O_1 \mid X_1, X_2)$*
- *8 additions for $\tau_2(X_3, O_1) = \sum_{X_1, X_2} \psi_2(X_1, X_2, X_3, O_1)$*

The same number of operations is required to eliminate X_3 and X_4. (Once these steps are completed, we can eliminate O_1, O_2 as usual.) Thus, the total number of operations required to eliminate all of $X_1, \ldots, X_4$ (after the elimination of A) is $2 \cdot (8 + 16) = 48$ multiplications and $2 \cdot 8 = 16$ additions, considerably more than our previous case. ■

Clearly, in the second network structure, had we done the decomposition of the noisy-or variable so as to make X_1 and X_3 parents of O_1 (and X_2, X_4 parents of O_2), we would get the same cost as we did in the first case. However, in order to do that, we need to take into consideration the global structure of the network, and even the order in which other variables are eliminated, at the same time that we are determining how to decompose a particular variable with symmetric ICI. In particular, we should determine the structure of the decomposition at the same time that we are considering the elimination ordering for the network as a whole.

9.6.1.4 Heterogeneous Factorization

An alternative approach that achieves this goal uses a different factorization for a network — one that factorizes the joint distribution for the network into CPDs, as well as the CPDs of symmetric ICI variables into smaller components. This factorization is *heterogeneous*, in that some factors must be combined by product, whereas others need to be combined using the type of operation that corresponds to the symmetric ICI function in the corresponding CPD. One can then define a heterogeneous variable elimination algorithm that combines factors, using whichever operation is appropriate, and that eliminates variables. Using this construction, we can determine a global ordering for the operations that determines the order in which both local

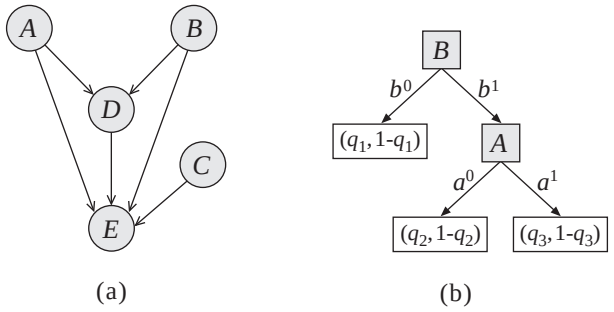


Figure 9.15 A Bayesian network with rule-based structure: (a) the network structure; (b) the CPD for the variable D.

factors and global factors are combined. Thus, in effect, the algorithm determines the order in which the components of an ICI CPD are "recombined" in a way that takes into consideration the structure of the factors created in a variable elimination algorithm.

9.6.2 Context-Specific Independence

A second important type of local CPD structure is the context-specific independence, typically encoded in a CPD as trees or rules. As in the case of ICI, there are two main ways of exploiting this type of structure in the context of a variable elimination algorithm. One approach (exercise 9.15) uses a decomposition of the CPD, which is performed as a preprocessing step on the network structure; standard variable elimination can then be performed on the modified network. The second approach, which we now describe, modifies the variable elimination algorithm itself to conduct its basic operations on structured factors. We can also exploit this structure within the context of a conditioning algorithm.

9.6.2.1 Rule-Based Variable Elimination

An alternative approach is to introduce the structure directly into the factors used in the variable elimination algorithm, allowing it to take advantage of the finer-grained structure. It turns out that this approach is easier to understand and implement for CPDs and factors represented as rules, and hence we present the algorithm in this context.

As specified in section 5.3.1.2, a rule-based CPD is described as a set of mutually exclusive and exhaustive rules, where each rule ρ has the form $\langle \boldsymbol{c}; p \rangle$. As we already discussed, a tree-CPD and a tabular CPD can each be converted into a set of rules in the obvious way.

Example 9.12

Consider the network structure shown in figure 9.15a. Assume that the CPD for the variable D is a tree, whose structure is shown in figure 9.15b. Decomposing this CPD into rules, we get the following

set of rules:

$$\left\{\begin{array}{ll} \rho_1 & \langle b^0, d^0; 1-q_1\rangle \\ \rho_2 & \langle b^0, d^1; q_1\rangle \\ \\ \rho_3 & \langle a^0, b^1, d^0; 1-q_2\rangle \\ \rho_4 & \langle a^0, b^1, d^1; q_2\rangle \\ \rho_5 & \langle a^1, b^1, d^0; 1-q_3\rangle \\ \rho_6 & \langle a^1, b^1, d^0; q_3\rangle \end{array}\right\}$$

Assume that the CPD $P(E \mid A, B, C, D)$ is also associated with a set of rules. Our discussion will focus on rules involving the variable D, so we show only that part of the rule set:

$$\left\{\begin{array}{ll} \rho_7 & \langle a^0, d^0, e^0; 1-p_1\rangle \\ \rho_8 & \langle a^0, d^0, e^1; p_1\rangle \\ \rho_9 & \langle a^0, d^1, e^0; 1-p_2\rangle \\ \rho_{10} & \langle a^0, d^1, e^1; p_2\rangle \\ \\ \rho_{11} & \langle a^1, b^0, c^1, d^0, e^0; 1-p_4\rangle \\ \rho_{12} & \langle a^1, b^0, c^1, d^0, e^1; p_4\rangle \\ \rho_{13} & \langle a^1, b^0, c^1, d^1, e^0; 1-p_5\rangle \\ \rho_{14} & \langle a^1, b^0, c^1, d^1, e^1; p_5\rangle \end{array}\right\}$$

■

Using this type of process, the entire distribution can be factorized into a multiset of rules $\mathcal{R}$, which is the union of all of the rules associated with the CPDs of the different variables in the network. Then, the probability of any instantiation ξ to the network variables $\mathcal{X}$ can be computed as

$$P(\xi) = \prod_{\langle \boldsymbol{c}; p\rangle \in \mathcal{R}, \xi \sim \boldsymbol{c}} p,$$

where we recall that $\xi \sim \boldsymbol{c}$ holds if the assignments ξ and $\boldsymbol{c}$ are compatible, in that they assign the same values to those variables that are assigned values in both.

Thus, as for the tabular CPDs, the distribution is defined in terms of a product of smaller components. In this case, however, we have broken up the tables into their component rows. This definition immediately suggests that we can use similar ideas to those used in the table-based variable elimination algorithm. In particular, we can multiply rules with each other and sum out a variable by adding up rules that give different values to the variables but are the same otherwise.

In general, we define the following two key operations:

Definition 9.8 rule product

Let $\rho_1 = \langle \boldsymbol{c}; p_1\rangle$ and $\rho_2 = \langle \boldsymbol{c}; p_2\rangle$ be two rules. Then their product $\rho_1 \cdot \rho_2 = \langle \boldsymbol{c}; p_1 \cdot p_2\rangle$. ■

This definition is significantly more restricted than the product of tabular factors, since it requires that the two rules have precisely the same context. We return to this issue in a moment.

Definition 9.9 rule sum

Let Y be a variable with $Val(Y) = \{y^1, \ldots, y^k\}$, and let ρ_i for $i = 1, \ldots, k$ be a rule of the form $\rho_i = \langle \boldsymbol{c}, Y = y^i; p_i\rangle$. Then for $\mathcal{R} = \{\rho_1, \ldots, \rho_k\}$, the sum $\sum_Y \mathcal{R} = \langle \boldsymbol{c}; \sum_{i=1}^k p_i\rangle$. ■

After this operation, Y is summed out in the context $\boldsymbol{c}$.

Both of these operations can only be applied in very restricted settings, that is, to sets of rules that satisfy certain stringent conditions. In order to make our set of rules amenable to the application of these operations, we might need to refine some of our rules. We therefore define the following final operation:

Definition 9.10 rule split

Let $\rho = \langle \boldsymbol{c}; p \rangle$ be a rule, and let Y be a variable. We define the rule split *$Split(\rho \angle Y)$ as follows: If $Y \in Scope[\boldsymbol{c}]$, then $Split(\rho \angle Y) = \{\rho\}$; otherwise,*

$$Split(\rho \angle Y) = \{\langle \boldsymbol{c}, Y = y; p \rangle \ : \ y \in Val(Y)\}.$$ ■

In general, the purpose of rule splitting is to make the context of one rule $\rho = \langle \boldsymbol{c}; p \rangle$ compatible with the context $\boldsymbol{c}'$ of another rule ρ'. Naively, we might take all the variables in $Scope[\boldsymbol{c}'] - Scope[\boldsymbol{c}]$ and split ρ recursively on each one of them. However, this process creates unnecessarily many rules.

Example 9.13

Consider ρ_2 and ρ_{14} in example 9.12, and assume we want to multiply them together. To do so, we need to split ρ_2 in order to produce a rule with an identical context. If we naively split ρ_2 on all three variables A, C, E that appear in ρ_{14} and not in ρ_2, the result would be eight rules of the form: $\langle a, b^0, c, d^1, e; q_1 \rangle$, one for each combination of values a, c, e. However, the only rule we really need in order to perform the rule product operation is $\langle a^1, b^0, c^1, d^1, e^1; q_1 \rangle$.

Intuitively, having split ρ_2 on the variable A, it is wasteful to continue splitting the rule whose context is a^0, since this rule (and any derived from it) will not participate in the desired rule product operation with ρ_{14}. Thus, a more parsimonious split of ρ_{14} that still generates this last rule is:

$$\left\{ \begin{array}{l} \langle a^0, b^0, d^1; q_1 \rangle \\ \langle a^1, b^0, c^0, d^1; q_1 \rangle \\ \langle a^1, b^0, c^1, d^1, e^0; q_1 \rangle \\ \langle a^1, b^0, c^1, d^1, e^1; q_1 \rangle \end{array} \right\}$$

This new rule set is still a mutually exclusive and exhaustive partition of the space originally covered by ρ_2, but contains only four rules rather than eight. ■

In general, we can construct these more parsimonious splits using the recursive procedure shown in algorithm 9.6. This procedure gives precisely the desired result shown in the example.

Rule splitting gives us the tool to take a set of rules and refine them, allowing us to apply either the rule-product operation or the rule-sum operation. The elimination algorithm is shown in algorithm 9.7. Note that the figure only shows the procedure for eliminating a single variable Y. The outer loop, which iteratively eliminates nonquery variables one at a time, is precisely the same as the Sum-Product-VE procedure in algorithm 9.1, except that it takes as input a set of rule factors rather than table factors.

To understand the operation of the algorithm more concretely, consider the following example:

Example 9.14

Consider the network in example 9.12, and assume that we want to eliminate D in this network. Our initial rule set $\mathcal{R}^+$ is the multiset of all of the rules whose scope contains D, which is precisely the set $\{\rho_1, \ldots, \rho_{14}\}$. Initially, none of the rules allows for the direct application of either rule product or rule sum. Hence, we have to split rules.

Algorithm 9.6 Rule splitting algorithm

```
    Procedure Rule-Split (
      ρ = ⟨c; p⟩,     // Rule to be split
      c'     // Context to split on
    )
1     if c ⊄ c' then return ρ
2     if Scope[c] ⊆ Scope[c'] then return ρ
3     Select Y ∈ Scope[c'] − Scope[c]
4     R ← Split(ρ∠Y)
5     R' ← ∪_{ρ''∈R} Rule-Split(ρ'', c')
6     return R'
```

The rules ρ_3 on the one hand, and ρ_7, ρ_8 on the other, have compatible contexts, so we can choose to combine them. We begin by splitting ρ_3 and ρ_7 on each other's context, which results in:

$$\left\{ \begin{array}{ll} \rho_{15} & \langle a^0, b^1, d^0, e^0; 1 - q_2 \rangle \\ \rho_{16} & \langle a^0, b^1, d^0, e^1; 1 - q_2 \rangle \\ \\ \rho_{17} & \langle a^0, b^0, d^0, e^0; 1 - p_1 \rangle \\ \rho_{18} & \langle a^0, b^1, d^0, e^0; 1 - p_1 \rangle \end{array} \right\}$$

The contexts of ρ_{15} and $\rho 18$ match, so we can now apply rule product, replacing the pair by:

$$\left\{ \begin{array}{ll} \rho_{19} & \langle a^0, b^1, d^0, e^0; (1 - q_2)(1 - p_1) \rangle \end{array} \right\}$$

We can now split ρ_8 using the context of ρ_{16} and multiply the matching rules together, obtaining

$$\left\{ \begin{array}{ll} \rho_{20} & \langle a^0, b^0, d^0, e^1; p_1 \rangle \\ \rho_{21} & \langle a^0, b^1, d^0, e^1; (1 - q_2) p_1 \rangle \end{array} \right\}.$$

The resulting rule set contains $\rho_{17}, \rho_{19}, \rho_{20}, \rho_{21}$ in place of ρ_3, ρ_7, ρ_8.

We can apply a similar process to ρ_4 and ρ_9, ρ_{10}, which leads to their substitution by the rule set:

$$\left\{ \begin{array}{ll} \rho_{22} & \langle a^0, b^0, d^1, e^0; 1 - p_2 \rangle \\ \rho_{23} & \langle a^0, b^1, d^1, e^0; q_2(1 - p_2) \rangle \\ \rho_{24} & \langle a^0, b^0, d^1, e^1; p_2 \rangle \\ \rho_{25} & \langle a^0, b^1, d^1, e^1; q_2 p_2 \rangle \end{array} \right\}.$$

We can now eliminate D in the context a^0, b^1, e^1. The only rules in $\mathcal{R}^+$ compatible with this context are ρ_{21} and ρ_{25}. We extract them from $\mathcal{R}^+$ and sum them; the resulting rule $\langle a^0, b^1, e^1; (1 - q_2)p_1 + q_2 p_2 \rangle$, is then inserted into $\mathcal{R}^-$. We can similarly eliminate D in the context a^0, b^1, e^0.

The process continues, with rules being split and multiplied. When D has been eliminated in a set of mutually exclusive and exhaustive contexts, then we have exhausted all rules involving D; at this point, $\mathcal{R}^+$ is empty, and the process of eliminating D terminates. ■

Algorithm 9.7 Sum-product variable elimination for sets of rules

Procedure Rule-Sum-Product-Eliminate-Var (
  $\mathcal{R}$, // Set of rules
  Y // Variable to be eliminated
)
1 $\mathcal{R}^+ \leftarrow \{\rho \in \mathcal{R} : \mathit{Scope}[\rho] \ni Y\}$
2 $\mathcal{R}^- \leftarrow \mathcal{R} - \mathcal{R}^+$
3 **while** $\mathcal{R}^+ \neq \emptyset$
4 Apply one of the following actions, when applicable
5 **Rule sum:**
6 Select $\mathcal{R}_{\boldsymbol{c}} \subseteq \mathcal{R}^+$ such that
7 $\mathcal{R}_{\boldsymbol{c}} = \{\langle \boldsymbol{c}, Y = y^1; p_1\rangle, \ldots, \langle \boldsymbol{c}, Y = y^k; p_k\rangle\}$
8 no other $\rho \in \mathcal{R}^+$ is compatible with $\boldsymbol{c}$)
9 $\mathcal{R}^- \leftarrow \mathcal{R}^- \cup \sum_Y \mathcal{R}_{\boldsymbol{c}}$
10 $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \mathcal{R}_{\boldsymbol{c}}$
11 **Rule product:**
12 Select $\langle \boldsymbol{c}; p_1\rangle, \langle \boldsymbol{c}; p_2\rangle \in \mathcal{R}^+$
13 $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \{\langle \boldsymbol{c}; p_1\rangle, \langle \boldsymbol{c}; p_2\rangle\} \cup \{\langle \boldsymbol{c}; p_1 \cdot p_2\rangle\}$
14 **Rule splitting for rule product:**
15 Select $\rho_1, \rho_2 \in \mathcal{R}^+$ such that
16 $\rho_1 = \langle \boldsymbol{c}_1; p_1\rangle$
17 $\rho_2 = \langle \boldsymbol{c}_2; p_2\rangle$
18 $\boldsymbol{c}_1 \sim \boldsymbol{c}_2$
19 $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \{\rho_1, \rho_2\} \cup \text{Rule-Split}(\rho_1, \boldsymbol{c}_2) \cup \text{Rule-Split}(\rho_2, \boldsymbol{c}_1)$
20 **Rule splitting for rule sum:**
21 Select $\rho_1, \rho_2 \in \mathcal{R}^+$ such that
22 $\rho_1 = \langle \boldsymbol{c}_1, Y = y^i; p_1\rangle$
23 $\rho_2 = \langle \boldsymbol{c}_2, Y = y^j; p_2\rangle$
24 $\boldsymbol{c}_1 \sim \boldsymbol{c}_2$
25 $i \neq j$
26 $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \{\rho_1, \rho_2\} \cup \text{Rule-Split}(\rho_1, \boldsymbol{c}_2) \cup \text{Rule-Split}(\rho_2, \boldsymbol{c}_1)$
27 **return** $\mathcal{R}^-$

A different way of understanding the algorithm is to consider its application to rule sets that originate from standard table-CPDs. It is not difficult to verify that the algorithm performs exactly the same set of operations as standard variable elimination. For example, the standard operation of factor product is simply the application of rule splitting on all of the rules that constitute the two tables, followed by a sequence of rule product operations on the resulting rule pairs. (See exercise 9.16.)

To prove that the algorithm computes the correct result, we need to show that each operation performed in the context of the algorithm maintains a certain correctness invariant. Let $\mathcal{R}$ be the current set of rules maintained by the algorithm, and $\boldsymbol{W}$ be the variables that have not yet been eliminated. Each operation must maintain the following condition:

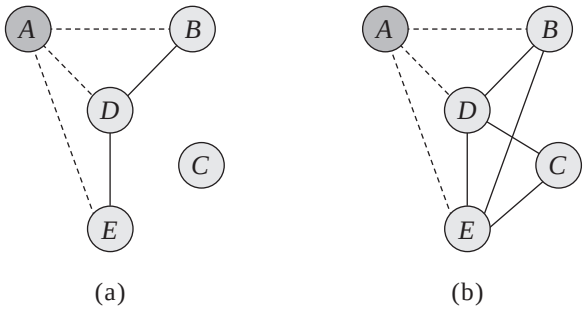


Figure 9.16 Conditioning a Bayesian network whose CPDs have CSI: (a) conditioning on a^0; (b) conditioning on a^1.

The probability of a context $\boldsymbol{c}$ such that $\mathit{Scope}[\boldsymbol{c}] \subseteq \boldsymbol{W}$ can be obtained by multiplying all rules $\langle \boldsymbol{c}'; p \rangle \in \mathcal{R}$ whose context is compatible with $\boldsymbol{c}$.

It is not difficult to show that the invariant holds initially, and that each step in the algorithm maintains it. Thus, the algorithm as a whole is correct.

9.6.2.2 Conditioning

We can also use other techniques for exploiting CSI in inference. In particular, we can generalize the notion of conditioning to this setting in an interesting way. Consider a network $\mathcal{B}$, and assume that we condition it on a variable U. So far, we have assumed that the structure of the different conditioned networks, for the different values u of U, is the same. When the CPDs are tables, with no extra structure, this assumption generally holds. However, when the CPDs have CSI, we might be able to utilize the additional structure to simplify the conditioned networks considerably.

Example 9.15 *Consider the network shown in figure 9.15, as described in example 9.12. Assume we condition this network on the variable A. If we condition on a^0, we see that the reduced CPD for E no longer depends on C. Thus, the conditioned Markov network for this set of factors is the one shown in figure 9.16a. By contrast, when we condition on a^1, the reduced factors do not "lose" any variables aside from A, and we obtain the conditioned Markov network shown in figure 9.16b. Note that the network in figure 9.16a is so simple that there is no point performing any further conditioning on it. Thus, we can continue the conditioning process for only one of the two branches of the computation — the one corresponding to a^1.* ■

In general, we can extend the conditioning algorithm of section 9.5 to account for CSI in the CPDs or in the factors of a Markov network. Consider a single conditioning step on a variable U. As we enumerate the different possible values u of U, we generate a possibly different conditioned network for each one. Depending on the structure of this network, we select which step to take next in the context of this particular network. In different networks, we might choose a different variable to use for the next conditioning step, or we might decide to stop the conditioning process for some networks altogether.

9.6.3 Discussion

We have presented two approaches to variable elimination in the case of local structure in the CPDs: preprocessing followed by standard variable elimination, and specialized variable elimination algorithms that use a factorization of the structured CPD. These approaches offer different trade-offs. On the one hand, the specialized variable elimination approach reveals more of the structure of the CPDs to the inference algorithm, allowing the algorithm more flexibility in exploiting this structure. Thus, this approach can achieve lower computational cost than any fixed decomposition scheme (see box 9.D). By comparison, the preprocessing approach embeds some of the structure within deterministic CPDs, a structure that most variable elimination algorithms do not fully exploit.

On the other hand, specialized variable elimination schemes such as those for rules require the use of special-purpose variable elimination algorithms rather than off-the-shelf packages. Furthermore, the data structures for tables are significantly more efficient than those for other types of factors such as rules. Although this difference seems to be an implementation issue, it turns out to be quite significant in practice. One can somewhat address this limitation by the use of more sophisticated algorithms that exploit efficient table-based operations whenever possible (see exercise 9.18).

☞ **Although the trade-offs between these two approaches is not always clear, it is generally the case that, in networks with significant amounts of local structure, it is valuable to design an inference scheme that exploits this structure for increased computational efficiency.**

Box 9.D — Case Study: Inference with Local Structure. *A natural question is the extent to which local structure can actually help speed up inference.*

In one experimental comparison by Zhang and Poole (1996), four algorithms were applied to fragments of the CPCS network (see box 5.D): standard variable elimination (with table representation of factors), the two decompositions illustrated in figure 9.14 for the case of noisy-or, and a special-purpose elimination algorithm that uses a heterogeneous factorization. The results show that in a network such as CPCS, which uses predominantly noisy-or and noisy-max CPDs, significant gains in performance can be obtained. They results also showed that the two decomposition schemes (tree-based and chain-based) are largely equivalent in their performance, and the heterogeneous factorization outperforms both of them, due to its greater flexibility in dynamically determining the elimination ordering during the course of the algorithm.

For rule-based variable elimination, no large networks with extensive rule-based structure had been constructed. So, Poole and Zhang (2003) used a standard benchmark network, with 32 variables and 11,018 entries. Entries that were within 0.05 *of each other were collaped, to construct a more compact rule-based representation, with a total of 5,834 distinct entries. As expected, there are a large number of cases where the use of rule-based inference provided significant savings. However, there were also many cases where contextual independence does not provide significant help, in which case the increased overhead of the rule-based inference dominates, and standard VE performs better.*

At a high level, the main conclusion is that table-based approaches are amenable to numerous optimizations, such as those described in box 10.A, which can improve the performance by an

order of magnitude or even more. Such optimizations are harder to define for more complex data structures. Thus, it is only useful to consider algorithms that exploit local structure either when it is extensively present in the model, or when it has specific structure that can, itself, be exploited using specialized algorithms.

9.7 Summary and Discussion

In this chapter, we described the basic algorithms for exact inference in graphical models. As we saw, probability queries essentially require that we sum out an exponentially large joint distribution. The fundamental idea that allows us to avoid the exponential blowup in this task is the use of dynamic programming, where we perform the summation of the joint distribution from the inside out rather than from the outside in, and cache the intermediate results, thereby avoiding repeated computation.

We presented an algorithm based on this insight, called variable elimination. The algorithm works using two fundamental operations over factors — multiplying factors and summing out variables in factors. We analyzed the computational complexity of this algorithm using the structural properties of the graph, showing that the key computational metric was the induced width of the graph.

We also presented another algorithm, called conditioning, which performs some of the summation operations from the outside in rather than from the inside out, and then uses variable elimination for the rest of the computation. Although the conditioning algorithm is never less expensive than variable elimination in terms of running time, it requires less storage space and hence provides a time-space trade-off for variable elimination.

We showed that both variable elimination and conditioning can take advantage of local structure within the CPDs. Specifically, we presented methods for making use of CPDs with independence of causal influence, and of CPDs with context-specific independence. In both cases, techniques tend to fall into two categories: In one class of methods, we modify the network structure, adding auxiliary variables that reveal some of the structure inside the CPD and break up large factors. In the other, we modify the variable elimination algorithm directly to use structured factors rather than tables.

Although exact inference is tractable for surprisingly many real-world graphical models, it is still limited by its worst-case exponential performance. There are many models that are simply too complex for exact inference. As one example, consider the $n \times n$ grid-structured pairwise Markov networks of box 4.A. It is not difficult to show that the minimal tree-width of this network is n. Because these networks are often used to model pixels in an image, where $n = 1,000$ is quite common, it is clear that exact inference is intractable for such networks. Another example is the family of networks that we obtain from the template model of example 6.11. Here, the moralized network, given the evidence, is a fully connected bipartite graph; if we have n variables on one side and m on the other, the minimal tree-width is $\min(n, m)$, which can be very large for many practical models. Although this example is obviously a toy domain, examples of similar structure arise often in practice. In later chapters, we will see many other examples where exact inference fails to scale up. Therefore, in chapter 11 and chapter 12 we

discuss approximate inference methods that trade off the accuracy of the results for the ability to scale up to much larger models.

One class of networks that poses great challenges to inference is the class of networks induced by template-based representations. These languages allow us to specify (or learn) very small, compact models, yet use them to construct arbitrarily large, and often densely connected, networks. Chapter 15 discusses some of the techniques that have been used to deal with dynamic Bayesian networks.

Our focus in this chapter has been on inference in networks involving only discrete variables. The introduction of continuous variables into the network also adds a significant challenge. Although the ideas that we described here are instrumental in constructing algorithms for this richer class of models, many additional ideas are required. We discuss the problems and the solutions in chapter 14.

9.8 Relevant Literature

The first formal analysis of the computational complexity of probabilistic inference in Bayesian networks is due to Cooper (1990).

peeling

forward-backward algorithm

nonserial dynamic programming

Variants of the variable elimination algorithm were invented independently in multiple communities. One early variant is the *peeling* algorithm of Cannings et al. (1976, 1978), formulated for the analysis of genetic pedigrees. Another early variant is the *forward-backward algorithm*, which performs inference in hidden Markov models (Rabiner and Juang 1986). An even earlier variant of this algorithm was proposed as early as 1880, in the context of continuous models (Thiele 1880). Interestingly, the first variable elimination algorithm for fully general models was invented as early as 1972 by Bertelé and Brioschi (1972), under the name *nonserial dynamic programming*. However, they did not present the algorithm in the setting of probabilistic inference in graph-structured models, and therefore it was many years before the connection to their work was recognized. Other early work with similar ideas but a very different application was done in the database community (Beeri et al. 1983).

The general problem of probabilistic inference in graphical models was first tackled by Kim and Pearl (1983), who proposed a local message passing algorithm in polytree-structured Bayesian networks. These ideas motivated the development of a wide variety of more general algorithms. One such trajectory includes the clique tree methods that we discuss at length in the next chapter (see also section 10.6). A second includes a specrum of other methods (for example, Shachter 1988; Shachter et al. 1990), culminating in the variable elimination algorithm, as presented here, first described by Zhang and Poole (1994) and subsequently by Dechter (1999). Huang and Darwiche (1996) provide some useful tips on an efficient implementation of algorithms of this type.

Dechter (1999) presents interesting connections between these algorithms and constraint-satisfaction algorithms, connections that have led to fruitful work in both communities. Other generalizations of the algorithm to settings other than pure probabilistic inference were described by Shenoy and Shafer (1990); Shafer and Shenoy (1990) and by Dawid (1992). The construction of the network polynomial was proposed by Darwiche (2003).

The complexity analysis of the variable elimination algorithm is described by Bertelé and Brioschi (1972); Dechter (1999). The analysis is based on core concepts in graph theory that have

been the subject of extensive theoretical analysis; see Golumbic (1980); Tarjan and Yannakakis (1984); Arnborg (1985) for an introduction to some of the key concepts and algorithms.

Much work has been done on the problem of finding low-tree-width triangulations or (equivalently) elimination orderings. One of the earliest algorithms is the maximum cardinality search of Tarjan and Yannakakis (1984). Arnborg, Corneil, and Proskurowski (1987) show that the problem of finding the minimal tree-width elimination ordering is $\mathcal{NP}$-hard. Shoikhet and Geiger (1997) describe a relatively efficient algorithm for finding this optimal elimination ordering — one whose cost is approximately the same as the cost of inference with the resulting ordering. Becker and Geiger (2001) present an algorithm that finds a close-to-optimal ordering. Nevertheless, most implementations use one of the standard heuristics. A good survey of these heuristic methods is presented by Kjærulff (1990), who also provides an extensive empirical comparison. Fishelson and Geiger (2003) suggest the use of stochastic search as a heuristic and provide another set of comprehensive experimental comparisons, focusing on the problem of genetic linkage analysis. Bodlaender, Koster, van den Eijkhof, and van der Gaag (2001) provide a series of simple preprocessing steps that can greatly reduce the cost of triangulation.

The first incarnation of the conditioning algorithm was presented by Pearl (1986a), in the context of cutset conditioning, where the conditioning variables cut all loops in the network, forming a polytree. Becker and Geiger (1994); Becker, Bar-Yehuda, and Geiger (1999) present a variety of algorithms for finding a small loop cutset. The general algorithm, under the name *global conditioning*, was presented by Shachter et al. (1994). They also demonstrated the equivalence of conditioning and variable elimination (or rather, the clique tree algorithm) in terms of the underlying computations, and pointed out the time-space trade-offs between these two approaches. These time-space trade-offs were then placed in a comprehensive computational framework in the *recursive conditioning* method of Darwiche (2001b); Allen and Darwiche (2003a,b). Cutset algorithms have made a significant impact on the application of genetic linkage analysis Schäffer (1996); Becker et al. (1998), which is particularly well suited to this type of method.

The two noisy-or decomposition methods were described by Olesen, Kjærulff, Jensen, Falck, Andreassen, and Andersen (1989) and Heckerman and Breese (1996). An alternative approach that utilizes a heterogeneous factorization was described by Zhang and Poole (1996); this approach is more flexible, but requires the use of a special-purpose inference algorithm. For the case of CPDs with context-specific independence, the decomposition approach was proposed by Boutilier, Friedman, Goldszmidt, and Koller (1996). The rule-based variable elimination algorithm was proposed by Poole and Zhang (2003). The trade-offs here are similar to the case of the noisy-or methods.

9.9 Exercises

Exercise 9.1⋆

Prove theorem 9.2.

Exercise 9.2⋆

Consider a factor produced as a product of some of the CPDs in a Bayesian network $\mathcal{B}$:

$$\tau(\boldsymbol{W}) = \prod_{i=1}^{k} P(Y_i \mid \mathrm{Pa}_{Y_i})$$

where $\boldsymbol{W} = \cup_{i=1}^{k}(\{Y_i\} \cup \mathrm{Pa}_{Y_i})$.

a. Show that τ is a conditional probability in some network. More precisely, construct another Bayesian network $\mathcal{B}'$ and a disjoint partition $\boldsymbol{W} = \boldsymbol{Y} \cup \boldsymbol{Z}$ such that $\tau(\boldsymbol{W}) = P_{\mathcal{B}'}(\boldsymbol{Y} \mid \boldsymbol{Z})$.
b. Conclude that all of the intermediate factors produced by the variable elimination algorithm are also conditional probabilities in some network.

Exercise 9.3

Consider a modified variable elimination algorithm that is allowed to multiply all of the entries in a single factor by some arbitrary constant. (For example, it may choose to renormalize a factor to sum to 1.) If we run this algorithm on the factors resulting from a Bayesian network with evidence, which types of queries can we still obtain the right answer to, and which not?

Exercise 9.4⋆

This exercise shows basic properties of the network polynomial and its derivatives:

a. Prove equation (9.8).
b. Prove equation (9.9).
c. Let $\boldsymbol{Y} = \boldsymbol{y}$ be some assignment. For $Y_i \in \boldsymbol{Y}$, we now consider what happens if we *retract* the observation $Y_i = y_i$. More precisely, let $\boldsymbol{y}_{-i}$ be the assignment in $\boldsymbol{y}$ to all variables other than Y_i. Show that

evidence retraction

$$\begin{aligned} P(\boldsymbol{y}_{-i}, Y_i = y_i' \mid \theta) &= \frac{\partial f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}^{\boldsymbol{y}})}{\lambda_{y_i'}} \\ P(\boldsymbol{y}_{-i} \mid \theta) &= \sum_{y_i'} \frac{\partial f_\Phi(\boldsymbol{\theta}, \boldsymbol{\lambda}^{\boldsymbol{y}})}{\lambda_{y_i'}}. \end{aligned}$$

Exercise 9.5⋆

In this exercise, you will show how you can use the gradient of the probability of a Bayesian network to perform *sensitivity analysis*, that is, to compute the effect on a probability query of changing the parameters in a single CPD $P(X \mid \boldsymbol{U})$. More precisely, let $\boldsymbol{\theta}$ be one set of parameters for a network $\mathcal{G}$, where we have that $\theta_{x|\boldsymbol{u}}$ is the parameter associated with the conditional probability entry $P(X \mid \boldsymbol{U})$. Let $\boldsymbol{\theta}'$ be another parameter assignment that is the same except that we replace the parameters $\boldsymbol{\theta}_{x|\boldsymbol{u}}$ with $\theta'_{x|\boldsymbol{u}} = \theta_{x|\boldsymbol{u}} + \Delta_{x|\boldsymbol{u}}$.

sensitivity analysis

For an assignment $\boldsymbol{e}$ (which may or may not involve variables in $X, \boldsymbol{U}$, compute the change $P(\boldsymbol{e} : \boldsymbol{\theta}) - P(\boldsymbol{e} : \boldsymbol{\theta}')$ in terms of $\Delta_{x|\boldsymbol{u}}$, and the network derivatives.

Exercise 9.6⋆

Consider some run of variable elimination over the factors Φ, where all variables are eliminated. This run generates some set of intermediate factors $\tau_i(\boldsymbol{W}_i)$. We can define a set of intermediate (arithmetic, not random) variables v_{ik} corresponding to the different entries $\tau_i(\boldsymbol{w}_i^k)$.

a. Show how, for each variable v_{ij}, we can write down an algebraic expression that defines v_{ij} in terms of: the parameters λ_{x_i}; the parameters $\theta_{\boldsymbol{x}_c}$; and variables v_{jl} for $j < i$.
b. Use your answer to the previous part to define an alternative representation whose complexity is linear in the total size of the intermediate factors in the VE run.
c. Show how the same representation can be used to compute all of the derivatives of the network polynomial; the complexity of your algorithm should be linear in the compact representation of the network polynomial that you derived in the previous part. (Hint: Consider the partial derivatives of the network polynomial relative to each v_{ij}, and use the chain rule for derivatives.)

Exercise 9.7

Prove proposition 9.1.

Exercise 9.8⋆

Prove theorem 9.10, by showing that any ordering produced by the maximum cardinality search algorithm eliminates cliques one by one, starting from the leaves of the clique tree.

Exercise 9.9

a. Show that variable elimination on polytrees can be performed in linear time, assuming that the local probability models are represented as full tables. Specifically, for any polytree, describe an elimination ordering, and show that the complexity of variable elimination with your ordering is linear in the size of the network. Note that the linear time bound here is in terms of the size of the CPTs in the network, so that the cost of the algorithm grows exponentially with the number of parents of a node.
b. Extend your result from (1) to apply to cases where the CPDs satisfy independence of causal influence. Note that, in this case, the network representation is linear in the number of variables in the network, and the algorithm should be linear in that number.
c. Now extend your result from (1) to apply to cases where the CPDs are tree-structured. In this case, the network representation is the sum of the sizes of the trees in the individual CPDs, and the algorithm should be linear in that number.

Exercise 9.10⋆

Consider the four criteria described in connection with Greedy-Ordering of algorithm 9.4: Min-Neighbors, Min-Weight, Min-Fill, and Weighted-Min-Fill. Show that none of these criteria dominate the others; that is, for any pair, there is always a graph where the ordering produced by one of them is better than that produced by the other. As our measure of performance, use the computational cost of full variable elimination (that is, for computing the partition function). For each counterexample, define the structure of the graph and the cardinality of the variables, and show the ordering produced by each member of the pair.

Exercise 9.11⋆

Let $\mathcal{H}$ be an undirected graph, and $\prec$ an elimination ordering. Prove that X—Y is a fill edge in the induced graph if and only if there is a path X—Z_1— . . . Z_k—Y in $\mathcal{H}$ such that $Z_i \prec X$ and $Z_i \prec Y$ for all $i = 1, \ldots, k$.

Exercise 9.12⋆

Prove theorem 9.12.

Exercise 9.13⋆

Prove theorem 9.13.

Exercise 9.14⋆

The standard conditioning algorithm first conditions the network on the conditioning variables $\boldsymbol{U}$, splitting the computation into a set of computations, one for every instantiation $\boldsymbol{u}$ to $\boldsymbol{U}$; it then performs variable elimination on the remaining network. As we discussed in section 9.5.4.1, we can generalize conditioning so that it alternates conditioning steps and elimination in an arbitrary way. In this question, you will formulate such an algorithm and provide a graph-theoretic analysis of its complexity.

Let Φ be a set of factors over $\mathcal{X}$, and let $\boldsymbol{X}$ be a set of nonquery variables. Define a *summation procedure* σ to be a sequence of operations, each of which is either *elim*(X) or *cond*(X) for some $X \in \boldsymbol{X}$, such that each $X \in \boldsymbol{X}$ appears in the sequence σ precisely once. The semantics of this procedure is that, going from left to right, we perform the operation described on the variables in sequence. For example, the summation procedure of example 9.5 would be written as:

$$\textit{elim}(A_{k-1}), \textit{elim}(A_{k-2}), \ldots \textit{elim}(A_1), \textit{cond}(A_k), \textit{elim}(C), \textit{elim}(B).$$

a. Define an algorithm that takes a summation sequence as input and performs the operations in the order stated. Provide precise pseudo-code for the algorithm.
b. Define the notion of an induced graph for this algorithm, and define the time and space complexity of the algorithm in terms of the induced graph.

Exercise 9.15⋆

In section 9.6.1.1, we described an approach to decomposing noisy-or CPDs, aimed at reducing the cost of variable elimination. In this exercise, we derive a construction for CPD-trees in a similar spirit.

a. Consider a variable Y that has a binary-valued parent A and four additional parents $X_1, \ldots, X_4$. Assume that the CPD of Y is structured as a tree whose first split is A, and where Y depends only on X_1, X_2 in the $A = a^1$ branch, and only on X_3, X_4 in the $A = a^0$ branch. Define two new variables, Y_{a^1} and Y_{a^0}, which represent the value that Y would take if A were to have the value a^1, and the value that Y would take if A were to have the value a^0. Define a new model for Y that is defined in terms of these new variables. Your model should precisely specify the CPDs for Y_{a^1}, Y_{a^0}, and Y in terms of Y's original CPD.
b. Define a general procedure that recursively decomposes a tree-CPD using the same principles.

Exercise 9.16

In this exercise, we show that rule-based variable elimination performs exactly the same operations as table-based variable elimination, when applied to rules generated from table-CPDs. Consider two table factors $\phi(\boldsymbol{X}), \phi'(\boldsymbol{Y})$. Let $\mathcal{R}$ be the set of constituent rules for $\phi(\boldsymbol{X})$ and $\mathcal{R}'$ the set of constituent rules for $\phi(\boldsymbol{Y})$.

a. Show that the operation of multiplying $\phi \cdot \phi'$ can be implemented as a series of rule splits on $\mathcal{R} \cup \mathcal{R}'$, followed by a series of rule products.
b. Show that the operation of summing out $Y \in \boldsymbol{X}$ in ϕ can be implemented as a series of rule sums in $\mathcal{R}$.

Exercise 9.17⋆

Prove that each step in the algorithm of algorithm 9.7 maintains the program-correctness invariant described in the text: Let $\mathcal{R}$ be the current set of rules maintained by the algorithm, and $\boldsymbol{W}$ be the variables that have not yet been eliminated. The invariant is that:

> The probability of a context $\boldsymbol{c}$ such that $\mathit{Scope}[\boldsymbol{c}] \subseteq \boldsymbol{W}$ can be obtained by multiplying all rules $\langle \boldsymbol{c}'; p \rangle \in \mathcal{R}$ whose context is compatible with $\boldsymbol{c}$.

Exercise 9.18⋆⋆

Consider an alternative factorization of a Bayesian network where each factor is a hybrid between a rule and a table, called a *confactor*. Like a rule, a confactor associated with a context $\boldsymbol{c}$; however, rather than a single number, each confactor contains not a single number, but a standard table-based factor. For example, the CPD of figure 5.4a would have a confactor, associated with the middle branch, whose context is a^1, s^0, and whose associated table is

l^0, j^0	0.9
l^0, j^1	0.1
l^1, j^0	0.4
l^1, j^1	0.6

Extend the rule splitting algorithm of algorithm 9.6 and the rule-based variable elimination algorithm of algorithm 9.7 to operate on confactors rather than rules. Your algorithm should use the efficient table-based data structures and operations when possible, resorting to the explicit partition of tables into rules only when absolutely necessary.

Exercise 9.19⋆⋆

We have shown that the sum-product variable elimination algorithm is sound, in that it returns the same answer as first multiplying all the factors, and then summing out the nonquery variables. Exercise 13.3 asks for a similar argument for max-product. One can prove similar results for other pairs of operations, such as max-sum. Rather than prove the same result for each pair of operations we encounter, we now provide a *generalized variable elimination* algorithm from which these special cases, as well as others, follow directly. This general algorithm is based on the following result, which is stated in terms of a pair of abstract operators: generalized combination of two factors, denoted $\phi_1 \otimes \phi_2$; and generalized marginalization of a factor ϕ over a subset $\boldsymbol{W}$, denoted $\Lambda_{\boldsymbol{W}}(\phi)$. We define our generalized variable elimination algorithm in direct analogy to the sum-product algorithm of algorithm 9.1, replacing factor product with $\otimes$ and summation for variable elimination with Λ.

generalized variable elimination

We now show that if these two operators satisfy certain conditions, the variable elimination algorithm for these two operations is sound:

Commutativity of combination: For any factors ϕ_1, ϕ_2:

$$\phi_1 \otimes \phi_2 = \phi_2 \otimes \phi_1. \tag{9.12}$$

Associativity of combination: For any factors ϕ_1, ϕ_2, ϕ_3:

$$\phi_1 \otimes (\phi_2 \otimes \phi_3) = (\phi_1 \otimes \phi_2) \otimes \phi_3. \tag{9.13}$$

Consonance of marginalization: If ϕ is a factor of scope $\boldsymbol{W}$, and $\boldsymbol{Y}, \boldsymbol{Z}$ are disjoint subsets of $\boldsymbol{W}$, then:

$$\Lambda_{\boldsymbol{Y}}(\Lambda_{\boldsymbol{Z}}(\phi)) = \Lambda_{(\boldsymbol{Y} \cup \boldsymbol{Z})}(\phi). \tag{9.14}$$

Marginalization over combination: If ϕ_1 is a factor of scope $\boldsymbol{W}$ and $\boldsymbol{Y} \cap \boldsymbol{W} = \emptyset$, then:

$$\Lambda_{\boldsymbol{Y}}(\phi_1 \otimes \phi_2) = \phi_1 \otimes \Lambda_{\boldsymbol{Y}}(\phi_2). \tag{9.15}$$

Show that if $\otimes$ and Λ satisfy the preceding axioms, then we obtain a theorem analogous to theorem 9.5. That is, the algorithm, when applied to a set of factors Φ and a set of variables to be eliminated $\boldsymbol{Z}$, returns a factor

$$\phi^*(\boldsymbol{Y}) = \Lambda_{\boldsymbol{Z}}(\bigotimes_{\phi \in \Phi} \phi).$$

Exercise 9.20⋆⋆

You are taking the final exam for a course on computational complexity theory. Being somewhat too theoretical, your professor has insidiously sneaked in some unsolvable problems and has told you that exactly K of the N problems have a solution. Out of generosity, the professor has also given you a probability distribution over the solvability of the N problems.

To formalize the scenario, let $\mathcal{X} = \{X_1, \ldots, X_N\}$ be binary-valued random variables corresponding to the N questions in the exam where $Val(X_i) = \{0(\text{unsolvable}), 1(\text{solvable})\}$. Furthermore, let $\mathcal{B}$ be a Bayesian network parameterizing a probability distribution over $\mathcal{X}$ (that is, problem i may be easily used to solve problem j so that the probabilities that i and j are solvable are not independent in general).

a. We begin by describing a method for computing the probability of a question being solvable. That is we want to compute $P(X_i = 1, \text{Possible}(\mathcal{X}) = K)$ where

$$\text{Possible}(\mathcal{X}) = \sum_i \mathbf{1}\{X_i = 1\}$$

is the number of solvable problems assigned by the professor.

To this end, we define an *extended factor* ϕ as a "regular" factor ψ and an index so that it defines a function $\phi(\boldsymbol{X}, L) : Val(\boldsymbol{X}) \times \{0, \ldots, N\} \mapsto \mathbb{R}$ where $\boldsymbol{X} = Scope[\phi]$. A projection of such a factor $[\phi]_l$ is a regular factor $\psi : Val(\boldsymbol{X}) \mapsto \mathbb{R}$, such that $\psi(\boldsymbol{X}) = \phi(\boldsymbol{X}, l)$.

Provide a definition of factor combination and factor marginalization for these extended factors such that

$$P(X_i, \text{Possible}(\mathcal{X}) = K) = \left[\sum_{\mathcal{X} - \{X_i\}} \prod_{\phi \in \Phi} \phi \right]_K , \tag{9.16}$$

where each $\phi \in \Phi$ is an extended factor corresponding to some CPD of the Bayesian network, defined as follows:

$$\phi_{X_i}(\{X_i\} \cup \mathbf{Pa}_{X_i}, k) = \begin{cases} P(X_i \mid \mathrm{Pa}_{X_i}) & \text{if } X_i = k \\ 0 & \text{otherwise} \end{cases}$$

b. Show that your operations satisfy the condition of exercise 9.19 so that you can compute equation (9.16) use the generalized variable elimination algorithm.

c. Realistically, you will have time to work on exactly M problems ($1 \leq M \leq N$). Obviously, your goal is to maximize the expected number of solvable problems that you attempt. (Luckily for you, every solvable problem that you attempt you will solve correctly, and you neither gain nor lose credit for working on an unsolvable problem.) Let $\boldsymbol{Y}$ be a subset of $\mathcal{X}$ indicating exactly M problems you choose to work on, and let

$$\text{Correct}(\mathcal{X}, \boldsymbol{Y}) = \sum_{X_i \in \boldsymbol{Y}} X_i$$

be the number of solvable problems that you attempt. The expected number of problems you solve is

$$\boldsymbol{E}_{P_{\mathcal{B}}}[\text{Correct}(\mathcal{X}, \boldsymbol{Y}) \mid \text{Possible}(\mathcal{X}) = K]. \tag{9.17}$$

Using your generalized variable elimination algorithm, provide an efficient algorithm for computing this expectation.

d. Your goal is to find $\boldsymbol{Y}$ that optimizes equation (9.17). Provide a simple example showing that:

$$\arg \max_{\boldsymbol{Y}:|\boldsymbol{Y}|=M} \boldsymbol{E}_{P_{\mathcal{B}}}[\text{Correct}(\mathcal{X}, \boldsymbol{Y})] \neq \arg \max_{\boldsymbol{Y}:|\boldsymbol{Y}|=M} \boldsymbol{E}_{P_{\mathcal{B}}}[\text{Correct}(\mathcal{X}, \boldsymbol{Y}) \mid \text{Possible}(\mathcal{X}) = K].$$

e. Give an efficient algorithm for finding

$$\arg \max_{\boldsymbol{Y}:|\boldsymbol{Y}|=M} \boldsymbol{E}_{P_{\mathcal{B}}}[\text{Correct}(\mathcal{X}, \boldsymbol{Y}) \mid \text{Possible}(\mathcal{X}) = K].$$

(Hint: Use linearity of expectations.)

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

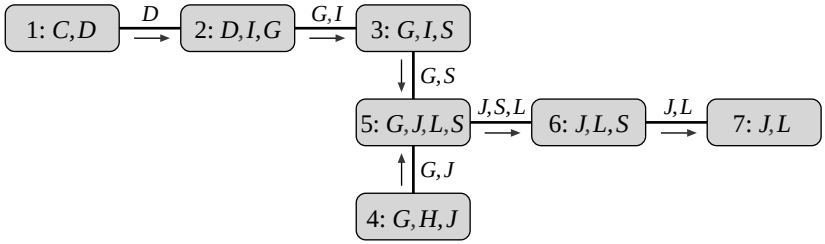


Figure 10.1 Cluster tree for the VE execution in table 9.1

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 $\boldsymbol{C}_i$, 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 message $\tau_3(G, S)$ 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 $\boldsymbol{C}_i$ is on the path from $\boldsymbol{C}_j$ to the root we say that $\boldsymbol{C}_i$ is *upstream* from $\boldsymbol{C}_j$, and $\boldsymbol{C}_j$ is *downstream* from $\boldsymbol{C}_i$. 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 $X \in \boldsymbol{C}_i$ and $X \in \boldsymbol{C}_j$, then X is also in every cluster in the (unique) path in $\mathcal{T}$ between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$.* ■

Note that the running intersection property implies that $\boldsymbol{S}_{i,j} = \boldsymbol{C}_i \cap \boldsymbol{C}_j$.

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 $\boldsymbol{C}_2$ and in $\boldsymbol{C}_4$, so it is also present in the cliques on the path between them: $\boldsymbol{C}_3$ and $\boldsymbol{C}_5$. ■

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 $\boldsymbol{C}$ and $\boldsymbol{C}'$ be two clusters that contain X. Let $\boldsymbol{C}_X$ 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 $\boldsymbol{C}$ and $\boldsymbol{C}_X$, and analogously for $\boldsymbol{C}'$, thereby proving the result.

First, we observe that the computation at $\boldsymbol{C}_X$ must take place later in the algorithm's execution than the computation at $\boldsymbol{C}$: When X is eliminated in $\boldsymbol{C}_X$, all of the factors involving X are multiplied into $\boldsymbol{C}_X$; 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 $\boldsymbol{C}$. We also know that X is not eliminated in $\boldsymbol{C}$. Therefore, the message computed in $\boldsymbol{C}$ must have X in its domain. By definition, the recipient of X's message, which is $\boldsymbol{C}$'s upstream neighbor in the tree, multiplies in the message

from $\boldsymbol{C}$. Hence, it will also have X in its scope. The same argument applies to show that all cliques upstream from $\boldsymbol{C}$ will have X in their scope, until X is eliminated, which happens only in $\boldsymbol{C}_X$. Thus, X must appear in all cliques between $\boldsymbol{C}$ and $\boldsymbol{C}_X$, 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 $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ be two neighboring clusters, such that $\boldsymbol{C}_i$ passes the message τ_i to $\boldsymbol{C}_j$. Then the scope of the message τ_i is precisely $\boldsymbol{C}_i \cap \boldsymbol{C}_j$.*

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 *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. ■

clique tree

clique

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 $\boldsymbol{S}_{i,j}$, let $\boldsymbol{W}_{<(i,j)}$ be the set of all variables in the scope of clusters in the $\boldsymbol{C}_i$ side of the tree, and $\boldsymbol{W}_{<(j,i)}$ be the set of all variables in the scope of clusters in the $\boldsymbol{C}_j$ side of the tree.

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

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

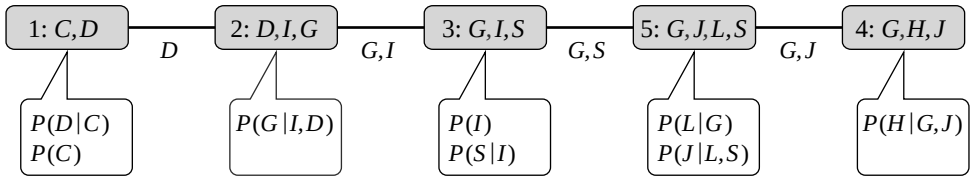


Figure 10.2 Simplified clique tree $\mathcal{T}$ for the Extended Student **network**

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:

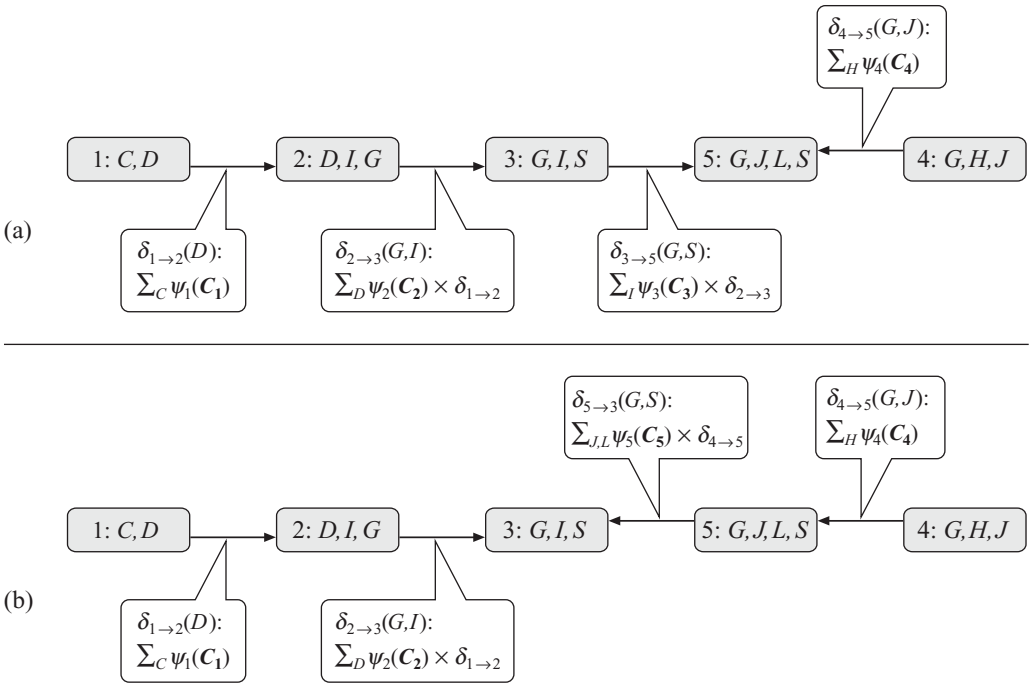


Figure 10.3 Two different message propagations with different root cliques in the Student **clique tree:** (a) $\boldsymbol{C}_5$ is the root; (b) $\boldsymbol{C}_3$ is the root.

1. **In $\boldsymbol{C}_1$:** We eliminate C by performing $\sum_C \psi_1(C, D)$. The resulting factor has scope D. We send it as a message $\delta_{1\to 2}(D)$ to $\boldsymbol{C}_2$.
2. **In $\boldsymbol{C}_2$:** We define $\beta_2(G, I, D) = \delta_{1\to 2}(D) \cdot \psi_2(G, I, D)$. We then eliminate D to get a factor over G, I. The resulting factor is $\delta_{2\to 3}(G, I)$, which is sent to $\boldsymbol{C}_3$.
3. **In $\boldsymbol{C}_3$:** We define $\beta_3(G, S, I) = \delta_{2\to 3}(G, I) \cdot \psi_3(G, S, I)$ and eliminate I to get a factor over G, S, which is $\delta_{3\to 5}(G, S)$.
4. **In $\boldsymbol{C}_4$:** We eliminate H by performing $\sum_H \psi_4(H, G, J)$ and send out the resulting factor as $\delta_{4\to 5}(G, J)$ to $\boldsymbol{C}_5$.
5. **In $\boldsymbol{C}_5$:** We define $\beta_5(G, J, S, L) = \delta_{3\to 5}(G, S) \cdot \delta_{4\to 5}(G, J) \cdot \psi_5(G, J, S, L)$.

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, $\boldsymbol{C}_4$ 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, $\boldsymbol{C}_2$ is ready only after it receives its message from $\boldsymbol{C}_1$. Thus, $\boldsymbol{C}_1, \boldsymbol{C}_4, \boldsymbol{C}_2, \boldsymbol{C}_3, \boldsymbol{C}_5$ is a legal execution ordering for a tree rooted at $\boldsymbol{C}_5$, whereas $\boldsymbol{C}_2, \boldsymbol{C}_1, \boldsymbol{C}_4, \boldsymbol{C}_3, \boldsymbol{C}_5$ 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 $\boldsymbol{C}_4$ as the root. Let us see how the algorithm would have changed in that case:

1. **In $\boldsymbol{C}_1$:** The computation and message are unchanged.
2. **In $\boldsymbol{C}_2$:** The computation and message are unchanged.
3. **In $\boldsymbol{C}_3$:** The computation and message are unchanged.
4. **In $\boldsymbol{C}_5$:** 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 $\boldsymbol{C}_4$.
5. **In $\boldsymbol{C}_4$:** 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 $\boldsymbol{C}_3$, for example, the computation in $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ is identical. The computation in $\boldsymbol{C}_4$ is the same as in the first of our two executions: a message is computed and sent to $\boldsymbol{C}_5$. In $\boldsymbol{C}_5$, we compute $\beta_5(G, J, S, L) = \delta_{4\to5}(G, J) \cdot \psi_5(G, J, S, L)$, and we eliminate J and L to produce a message $\delta_{5\to3}(G, S)$, which can then be sent to $\boldsymbol{C}_3$ 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 $\boldsymbol{C}_1, \ldots, \boldsymbol{C}_k$. 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 $\boldsymbol{C}_j$ to be:

$$\psi_j(\boldsymbol{C}_j) = \prod_{\phi \,:\, \alpha(\phi)=j} \phi.$$

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

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

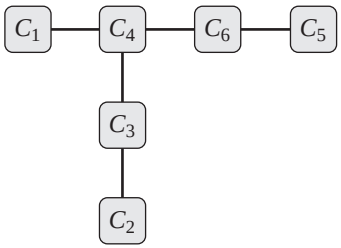


Figure 10.4 An abstract clique tree that is not chain-structured

Let $\boldsymbol{C}_r$ 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 $\boldsymbol{C}_i$, we define Nb_i to be the set of indexes of cliques that are neighbors of $\boldsymbol{C}_i$. Let $p_r(i)$ be the upstream neighbor of i (the one on the path to the root clique r). Each clique $\boldsymbol{C}_i$, except for the root, performs a message passing computation and sends a message to its upstream neighbor $\boldsymbol{C}_{p_r(i)}$.

sum-product message passing

The message from $\boldsymbol{C}_i$ to another clique $\boldsymbol{C}_j$ is computed using the following *sum-product message passing* computation:

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

In words, the clique $\boldsymbol{C}_i$ 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 $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$, and sends the resulting factor as a message to $\boldsymbol{C}_j$.

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 $\beta_r(\boldsymbol{C}_r)$. It represents, as we show,

beliefs

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

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 $\boldsymbol{C}_6$ as our root clique. The numbering of the cliques denotes one possible ordering of the operations, with $\boldsymbol{C}_1$ 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 $\boldsymbol{Y}$ which is fully contained in some clique. We select one such clique $\boldsymbol{C}_r$ to be the root, and perform the clique-tree message passing toward that root. We then extract $\tilde{P}_\Phi(\boldsymbol{Y})$ from the final potential at $\boldsymbol{C}_r$ by summing out the other variables $\boldsymbol{C}_r - \boldsymbol{Y}$.

Algorithm 10.1 Upward pass of variable elimination in clique tree

```
    Procedure CTree-SP-Upward (
      Φ,    // Set of factors
      T,    // Clique tree over Φ
      α,    // 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       δ_{i→p_r(i)}(S_{i,p_r(i)}) ← SP-Message(i, p_r(i))
5     β_r ← ψ_r · ∏_{k∈Nb_{C_r}} δ_{k→r}
6     return β_r

    Procedure Initialize-Cliques (

    )
1     for each clique C_i
2       ψ_i(C_i) ← ∏_{φ_j : α(φ_j)=i} φ_j
3

    Procedure SP-Message (
      i,    // sending clique
      j     // receiving clique
    )
1     ψ(C_i) ← ψ_i · ∏_{k∈(Nb_i−{j})} δ_{k→i}
2     τ(S_{i,j}) ← ∑_{C_i−S_{i,j}} ψ(C_i)
3     return τ(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 $\boldsymbol{C}_i$ to a neighboring $\boldsymbol{C}_j$ such that $X \in \boldsymbol{C}_i$ and $X \notin \boldsymbol{C}_j$. We first prove the following result:

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

Proof The proof is a simple consequence of the running intersection property. Assume by contradiction that X appears in some other clique $\boldsymbol{C}_k$ that is on the $\boldsymbol{C}_j$ side of the tree. Then $\boldsymbol{C}_j$ is on the path from $\boldsymbol{C}_i$ to $\boldsymbol{C}_k$. But we know that X appears in both $\boldsymbol{C}_i$ and $\boldsymbol{C}_k$ but not in $\boldsymbol{C}_j$, violating the running intersection property. ■

Based on this result, we can provide a semantic interpretation for the messages used in the clique tree. Let $(i\text{-}j)$ be some edge in the clique tree. We use $\mathcal{F}_{\prec(i\to j)}$ to denote the set of factors in the cliques on the $\boldsymbol{C}_i$-side of the edge and $\mathcal{V}_{\prec(i\to j)}$ 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 $\mathcal{F}_{\prec(3\to 5)} = \{P(C), P(D \mid C), P(G \mid I, D), P(I), P(S \mid I)\}$ and $\mathcal{V}_{\prec(3\to 5)} = \{C, D, I\}$. Intuitively, the message passed between the cliques $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ is the product of all the factors in $\mathcal{F}_{\prec(i\to j)}$, marginalized over the variables in the sepset (that is, summing out all the others).

Theorem 10.3

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

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

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

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

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 $\mathcal{V}_{\prec(i\to j)}$ is the disjoint union of $\mathcal{V}_{\prec(i_k\to i)}$ for $k = 1, \ldots, m$ and the variables $\boldsymbol{Y}_i$ eliminated at $\boldsymbol{C}_i$ itself. Similarly, $\mathcal{F}_{\prec(i\to j)}$ is the disjoint union of the $\mathcal{F}_{\prec(i_k\to i)}$ and the factors $\mathcal{F}_i$ from which ψ_i was computed. Thus equation (10.3) is equal to

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

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

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

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

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

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

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 $\boldsymbol{C}_r$ be the root clique in a clique tree, and assume that β_r is computed as in the algorithm of algorithm 10.1. Then*

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

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 $\boldsymbol{e}$, then $\beta_r(\boldsymbol{C}_r) = P_\mathcal{B}(\boldsymbol{C}_r, \boldsymbol{e})$. For a Markov network $\mathcal{H}$, if Φ consists of the compatibility functions defining the network, then $\beta_r(\boldsymbol{C}_r) = \tilde{P}_\Phi(\boldsymbol{C}_r)$. In both cases, we can obtain the probability over the variables in $\boldsymbol{C}_r$ 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 $\beta_r(\boldsymbol{C}_r)$.

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 $\boldsymbol{C}_5$ is the root, one where $\boldsymbol{C}_4$ is the root, and one where $\boldsymbol{C}_3$ is the root. As we pointed out, the messages sent from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ and from $\boldsymbol{C}_2$ to $\boldsymbol{C}_3$ are the same in all three executions. The message sent from $\boldsymbol{C}_4$ to $\boldsymbol{C}_5$ is the same in both of the executions where it appears. In the second of the three executions, there simply is no message from $\boldsymbol{C}_4$ to $\boldsymbol{C}_5$ — the message goes the other way, from $\boldsymbol{C}_5$ to $\boldsymbol{C}_4$.

More generally, consider two neighboring cliques $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ in some clique tree. It follows from theorem 10.3 that the value of the message sent from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$ does not depend on specific choice of root clique: As long as the root clique is on the $\boldsymbol{C}_j$-side, exactly the same message is sent from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$. The same argument applies if the root is on the $\boldsymbol{C}_i$-side. Thus, in all executions of the clique tree algorithm, whenever a message is sent between two cliques in the same direction, 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.

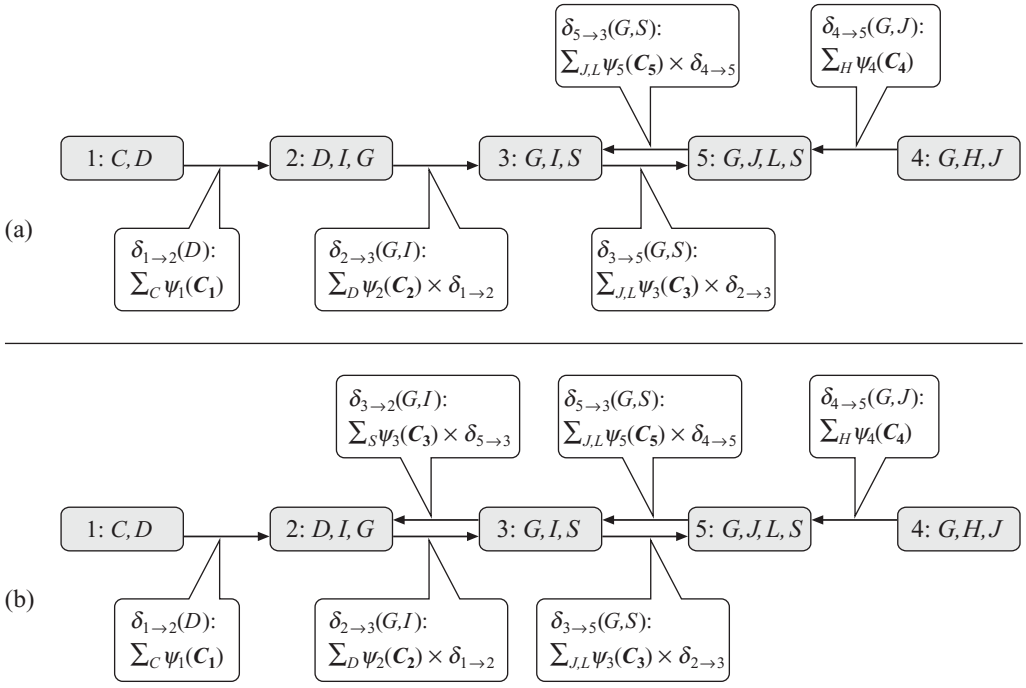


Figure 10.5 Two steps in a downward pass in the Student **network**

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 $\mathbf{C}_i$ is ready *to transmit to a neighbor $\mathbf{C}_j$ when $\mathbf{C}_i$ has messages from all of its neighbors except from $\mathbf{C}_j$.* ■

When $\mathbf{C}_i$ is ready to transmit to $\mathbf{C}_j$, it can compute the message $\delta_{i \to j}(\mathbf{S}_{i,j})$ by multiplying its initial potential with all of its incoming messages except the one from $\mathbf{C}_j$, and then eliminate the variables in $\mathbf{C}_i - \mathbf{S}_{i,j}$. 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 (
  Φ,    // Set of factors
  T     // Clique tree over Φ
)
1   Initialize-Cliques
2   while exist i, j such that i is ready to transmit to j
3     δ_{i→j}(S_{i,j}) ← SP-Message(i, j)
4   for each clique i
5     β_i ← ψ_i · ∏_{k∈Nb_i} δ_{k→i}
6   return {β_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 $\boldsymbol{C}_5$ is the root. Figure 10.5a shows a possible first step in a downward pass, where $\boldsymbol{C}_5$ sends a message to its child $\boldsymbol{C}_3$, based on the message from $\boldsymbol{C}_4$ 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 $\boldsymbol{C}_3$ sending the downward message to $\boldsymbol{C}_2$. ■

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 $\boldsymbol{C}_i$ 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

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

Note that it is important that $\boldsymbol{C}_i$ compute the message to a neighboring clique $\boldsymbol{C}_j$ 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 $\boldsymbol{C}_j$ (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 $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ are said to be calibrated *if*

$$\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{C}_i) = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j(\boldsymbol{C}_j).$$

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 $\beta_i(\boldsymbol{C}_i)$ and* sseptet beliefs *for*

beliefs

$$\mu_{i,j}(\boldsymbol{S}_{i,j}) = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{C}_i) = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j(\boldsymbol{C}_j). \tag{10.7}$$

■

☞ **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

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

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 vari-
         ables
)
1    j ← 0, k ← 0
2    for l = 0, ..., |X_1 ∪ X_2|
3      assignment[l] ← 0
4    for i = 0, ..., |Val(X_1 ∪ X_2)| − 1
5      psi[i] ← phi1[j] · phi2[k]
6      for l = 0, ..., |X_1 ∪ X_2|
7        assignment[l] ← assignment[l] + 1
8        if assignment[l] = card[l] then
9          assignment[l] ← 0
10           j ← j − (card[l] − 1) · phi1.stride[l]
11           k ← k − (card[l] − 1) · phi2.stride[l]
12         else
13           j ← j + phi1.stride[l]
14           k ← k + phi2.stride[l]
15           break
16   return (psi)
```

and vice versa

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

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* $\texttt{phi.stride}[X] = 0$ *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 large 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:

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

We also have that:

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

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:

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

Assignment			Marg_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		$\text{Marg}_{A,C}$
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			Marg_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:

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

■

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:*

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

where

$$\mu_{i,j} = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{C}_i) = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j(\boldsymbol{C}_j).$$

■

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 $\beta_i(\boldsymbol{C}_i)$ 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 $\beta_i(\boldsymbol{C}_i) \propto \tilde{P}_\Phi(\boldsymbol{C}_i)$.

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

$$\tilde{P}_\Phi \models (\boldsymbol{C}_i \perp \boldsymbol{X} \mid \boldsymbol{S}_{i,p_r(i)}).$$

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

$$\tilde{P}_\Phi(\mathcal{X}) = \tilde{P}_\Phi(\boldsymbol{C}_r) \cdot \prod_{i \neq r} \tilde{P}_\Phi(\boldsymbol{C}_i \mid \boldsymbol{S}_{i,p_r(i)}).$$

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

$$Q_{\mathcal{T}}(\mathcal{X}) = \beta_r(\boldsymbol{C}_r) \cdot \prod_{i \neq r} \beta_i(\boldsymbol{C}_i \mid \boldsymbol{S}_{i,p_r(i)}).$$

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:

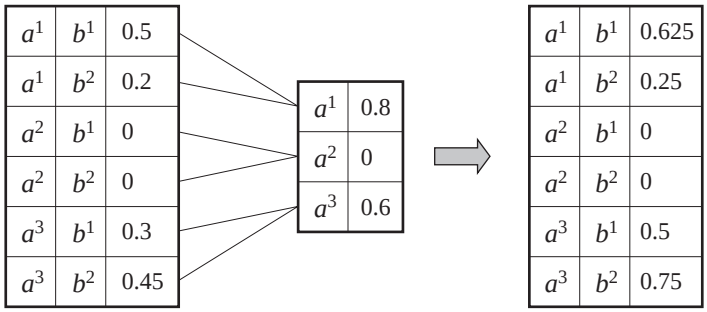


Figure 10.7 An example of factor division

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:

$$\delta_{i \to j} = \frac{\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i}{\delta_{j \to i}}. \tag{10.13}$$

Example 10.7

*Let us return to the simple clique tree in example 10.5, and assume that $\boldsymbol{C}_2$ serves as the (de facto) root, so that we first pass messages from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ and from $\boldsymbol{C}_3$ to $\boldsymbol{C}_2$. 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 $\boldsymbol{C}_i$ 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 $\boldsymbol{C}_i$ initializes β_i as ψ_i and then updates it by multiplying with message updates received from its neighbors. Each sepset $\boldsymbol{S}_{i,j}$ 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 $\boldsymbol{C}_2$ initially passes an uninformed message to $\boldsymbol{C}_3$: $\sigma_{2 \to 3} = \sum_B \psi_2(B, C)$. This message multiplies the beliefs about $\boldsymbol{C}_3$, 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 $\boldsymbol{C}_3$ sends a message to $\boldsymbol{C}_2$: $\sigma_{3 \to 2}(C) = \sum_D \beta_3(C, D)$. This message is divided by $\mu_{2,3}$, so the actual update for $\boldsymbol{C}_2$ 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 $\boldsymbol{C}_2$ would have received from $\boldsymbol{C}_3$ in the case where $\boldsymbol{C}_2$ 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 $\boldsymbol{C}_2$ receives a message from $\boldsymbol{C}_1$, containing $\sum_A \psi_1(A, B)$. The sepset $\boldsymbol{S}_{1,2}$ contains a message that is identically 1, so that this message is transmitted as is. At this point, $\boldsymbol{C}_2$ 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 (
      Φ,    // Set of factors
      T     // Clique tree over Φ
    )
1     Initialize-CTree
2     while exists an uninformed clique in T
3       Select (i–j) ∈ E_T
4       BU-Message(i, j)
5     return {β_i}

    Procedure Initialize-CTree (
    )
1     for each clique C_i
2       β_i ← ∏_{φ : α(φ)=i} φ
3     for each edge (i–j) ∈ E_T
4       μ_{i,j} ← 1

    Procedure BU-Message (
      i,    // sending clique
      j     // receiving clique
    )
1     σ_{i→j} ← ∑_{C_i − S_{i,j}} β_i
2       // marginalize the clique over the sepset
3     β_j ← β_j · σ_{i→j} / μ_{i,j}
4     μ_{i,j} ← σ_{i→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 $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ between which we will pass a message. Interestingly, we can make this choice arbitrarily, without damaging the correctness of the algorithm. For example, if $\boldsymbol{C}_i$ (for some reason) passes the same message to $\boldsymbol{C}_j$ 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 $\boldsymbol{C}_i$ passes a message to $\boldsymbol{C}_j$ 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:

$$\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i = \mu_{i,j} = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j.$$

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 $\boldsymbol{C}_i, \boldsymbol{C}_j$, 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 $\{\beta'_i : \boldsymbol{C}_i \in \mathcal{T}\}$, $\{\mu'_{i,j} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}\}$, 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 $\delta'_{i \to j}$, β'_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 $\boldsymbol{C}_r$.

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

incremental update

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.

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 $\mathbb{1}\{Z = z\}$. 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,

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

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

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

Thus, we obtain a representation of $\tilde{P}'_\Phi$ in the clique tree simply by multiplying in the new factor $\mathbf{1}\{Z = z\}$ 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 $P(\boldsymbol{Y} \mid \boldsymbol{e})$ 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 Φ
  $\{\beta_i\}, \{\mu_{i,j}\}$, // Calibrated clique and sepset beliefs for $\mathcal{T}$
  $\boldsymbol{Y}$ // A query
)
1 Let $\mathcal{T}'$ be a subtree of $\mathcal{T}$ such that $\boldsymbol{Y} \subseteq \mathit{Scope}[\mathcal{T}']$
2 Select a clique $r \in \mathcal{V}_{\mathcal{T}'}$ to be the root
3 $\Phi \leftarrow \beta_r$
4 **for** each $i \in \mathcal{V}_{\mathcal{T}'} - \{r\}$
5 $\phi \leftarrow \frac{\beta_i}{\mu_{i,p_r(i)}}$
6 $\Phi \leftarrow \Phi \cup \{\phi\}$
7 $\boldsymbol{Z} \leftarrow \mathit{Scope}[\mathcal{T}'] - \boldsymbol{Y}$
8 Let $\prec$ be some ordering over $\boldsymbol{Z}$
9 **return** Sum-Product-VE($\Phi, \boldsymbol{Z}, \prec$)

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 $\boldsymbol{C}_i, \boldsymbol{C}_j$ that contains $\tilde{P}_\Phi(\boldsymbol{C}_i, \boldsymbol{C}_j)$. We construct the table for i, j in order of the distance between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ in the tree.

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

$$\tilde{P}_\Phi(\boldsymbol{C}_j \mid \boldsymbol{C}_i) = \frac{\beta_j(\boldsymbol{C}_j)}{\mu_{i,j}(\boldsymbol{C}_i \cap \boldsymbol{C}_j)}.$$

From these, we can compute $\tilde{P}_\Phi(\boldsymbol{C}_i, \boldsymbol{C}_j)$.

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

$$\tilde{P}_\Phi \models (\boldsymbol{C}_i \perp \boldsymbol{C}_j \mid \boldsymbol{C}_l).$$

Thus, we can compute

$$\tilde{P}_\Phi(\boldsymbol{C}_i, \boldsymbol{C}_j) = \sum_{\boldsymbol{C}_l - \boldsymbol{C}_j} \tilde{P}_\Phi(\boldsymbol{C}_i, \boldsymbol{C}_l) \tilde{P}_\Phi(\boldsymbol{C}_j \mid \boldsymbol{C}_l),$$

where $\tilde{P}_\Phi(\boldsymbol{C}_j \mid \boldsymbol{C}_l)$ can be easily computed from the marginal $\tilde{P}_\Phi(\boldsymbol{C}_j, \boldsymbol{C}_l)$.

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 $\boldsymbol{C}_i$ corresponds to the factor ψ_i generated during the execution of the algorithm, and an undirected edge connects $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ 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 $\boldsymbol{C}_j$ which is a strict subset of some other cluster $\boldsymbol{C}_i$:

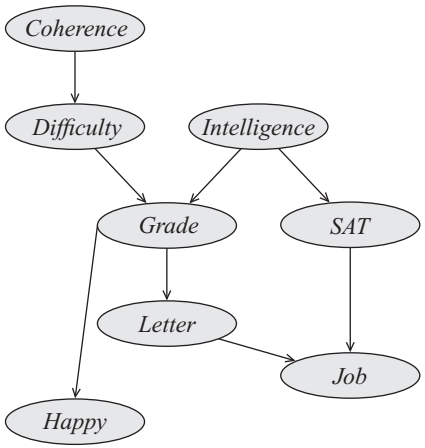


Figure 10.8 A modified Student **BN with an unambitious student**

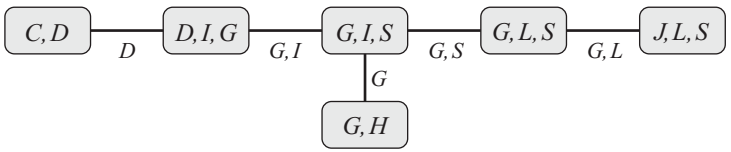


Figure 10.9 A clique tree for the modified Student **BN of figure 10.8**

Theorem 10.6 *Let $\mathcal{T}$ be a clique tree for a set of factors Φ. Then there exists a clique tree $\mathcal{T}'$ such that:*

- *each clique in $\mathcal{T}'$ is also a clique in $\mathcal{T}$;*
- *there is no pair of cliques $\boldsymbol{C}_i, \boldsymbol{C}_j$ in $\mathcal{T}'$ such that $\boldsymbol{C}_j \subset \boldsymbol{C}_i$.*

Proof The proof is constructive, eliminating redundant cliques one by one. We begin with $\mathcal{T}' = \mathcal{T}$. Let $\boldsymbol{C}_j, \boldsymbol{C}_i$ be a pair of cliques in $\mathcal{T}'$ such that $\boldsymbol{C}_j \subset \boldsymbol{C}_i$. By the running intersection property, $\boldsymbol{C}_j$ is a subset of all cliques on the path between $\boldsymbol{C}_j$ and $\boldsymbol{C}_i$. Let $\boldsymbol{C}_l$ be some neighbor clique of $\boldsymbol{C}_j$ such that $\boldsymbol{C}_j \subseteq \boldsymbol{C}_l$. We simply remove $\boldsymbol{C}_j$ from the tree, and connect all of the neighbors of $\boldsymbol{C}_j$, except for $\boldsymbol{C}_l$ itself, directly to $\boldsymbol{C}_l$. The proof that the resulting structure is a valid clique tree is not difficult, and it is left as an exercise (exercise 10.16).

As each of the clique elimination steps reduces the number of cliques in the tree, the process must terminate. When it does, we have a valid clique tree that does not contain a pair of cliques where one subsumes the other. ■

The reduction used in this theorem is precisely the one we used to transform the tree in figure 10.1 to the one in figure 10.2. Consider also the following slightly more complex example (one that does not result in a clique tree that has the form of a chain):

Example 10.10 *Assume that our student plans to be a beach bum upon graduation, so his happiness does not depend on getting a job. On the other hand, his happiness still depends on his grade. The network is shown in figure 10.8. Variable elimination with the ordering J, L, S, H, C, D, I, G, followed by a pruning of the nonmaximal clusters from the tree (as in theorem 10.6), we obtain the clique tree shown in figure 10.9.* ■

10.4.2 Clique Trees from Chordal Graphs

Theorem 10.6 shows that there exists a clique tree for Φ whose cliques are precisely the maximal cliques in $\mathcal{I}_{\Phi,\prec}$. This observation leads us to an alternative approach to constructing a clique tree. As we discussed in section 9.4.3.1, the induced graph $\mathcal{I}_{\prec,\Phi}$ is necessarily a chordal graph. In fact, the converse also holds: any chordal graph can be used as the basis for inference. To see that, recall that theorem 4.12 states that any chordal graph is associated with a clique tree. The algorithms presented in this chapter show that any clique tree can be used for inference. Thus, for any set of factors Φ, we can construct a clique tree for inference over Φ by constructing a chordal graph $\mathcal{H}^*$ that contains the edges in $\mathcal{H}_\Phi$, finding the maximal cliques in it, and connecting them appropriately. We now discuss each of these steps.

triangulation

The process of constructing a chordal graph that subsumes an existing graph $\mathcal{H}$ is called *triangulation*. Not surprisingly, finding a minimum triangulation, that is, one where the largest clique in the resulting chordal graph has minimum size, is $\mathcal{NP}$-hard. There are exact algorithms for finding an optimal triangulation of a graph, which are exponential in the size of the largest clique in the graph. In practice, these algorithms are too expensive, and one typically resorts to heuristic algorithms. Other triangulation methods provide a guarantee that the size of the largest clique is within a certain constant factor of optimal. These algorithms are less expensive, but they are still typically too costly in most applications. In practice, the most common approach to triangulation in graphical models uses the node-elimination techniques that we discussed in section 9.4.3.2.

Given a chordal graph $\mathcal{H}$, we now must find the maximal cliques in the graph. In general, finding the maximal cliques in a graph is also an $\mathcal{NP}$-hard problem. However, for chordal graphs the problem is quite easy. There are several methods. One of the simplest is to run maximum cardinality search on the resulting chordal graph and collect the maximal cliques generated in the process. By theorem 9.10, this process introduces no fill edges. It follows from theorem 9.6 that this process therefore generates all of the maximal cliques in $\mathcal{H}$. Another method is to begin with a family, each member of which is guaranteed to be a clique, and then use a greedy algorithm that adds nodes to the clique until it no longer induces a fully connected subgraph. This algorithm can be performed efficiently, because the number of maximal cliques in a chordal graph is at most n.

Finally, we must determine the edges in the clique tree. Again, one approach for achieving this task is maximum cardinality search, which also dictates which clique transmits information to which other clique. A somewhat more efficient approach that achieves the same effect is via a *maximum spanning tree* algorithm. More specifically, we build an undirected graph whose nodes are the maximal cliques in $\mathcal{H}$, and where every pair of nodes $\boldsymbol{C}_i, \boldsymbol{C}_j$ is connected by an edge whose *weight* is $|\boldsymbol{C}_i \cap \boldsymbol{C}_j|$. We then use a standard algorithm to find a tree in this graph whose weight — that is, the sum of the weights of the edges in the graph — is maximal.

maximum spanning tree

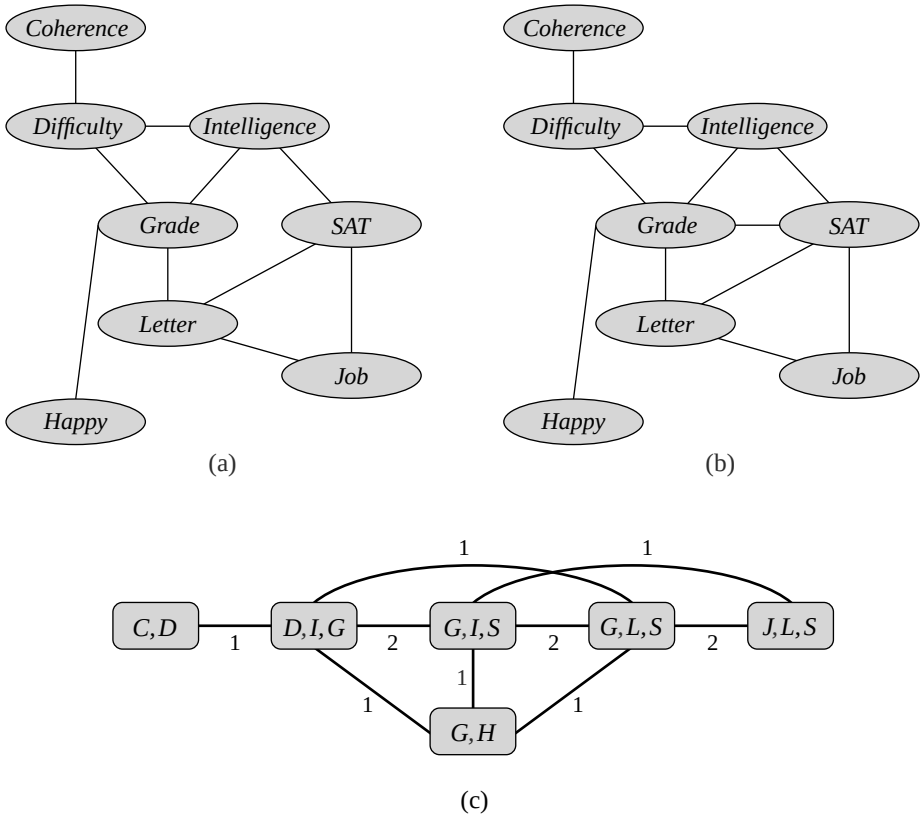


Figure 10.10 Example of a clique-tree construction algorithm: (a) Undirected factor graph (moralized graph for the network of figure 10.8). (b) One possible triangulation for this graph. (c) Cluster graph with edge weights where clusters are the cliques.

One such algorithm is shown in algorithm A.2. We can show that this approach results in the same structure as the one constructed by maximal cardinality search, and therefore it satisfies the running intersection property (see exercise 10.17).

To summarize, we can construct a clique tree as follows:

1. Given a set of factors, construct the undirected graph $\mathcal{H}_\Phi$.
2. Triangulate $\mathcal{H}_\Phi$ to construct a chordal graph $\mathcal{H}^*$.
3. Find cliques in $\mathcal{H}^*$, and make each one a node in a cluster graph.
4. Run the maximum spanning tree algorithm on the cluster graph to construct a tree.

Example 10.11

Consider again the network of figure 10.8. The undirected graph $\mathcal{H}_\Phi$ for this example is simply the moralized graph, shown in figure 10.10a. One possible triangulation for this graph, shown in

figure 10.10b, is obtained by introducing the fill edge G—S. This triangulation is a minimal one, as is the one that introduces the edge L, I. There are also, of course, other nonminimal triangulations. The graph of the maximal cliques for the chordal graph of figure 10.10b, with the associated weights on the edges, is shown in figure 10.10c. It is easy to verify that the maximum weight spanning tree is the clique tree shown in figure 10.9. ■

Once a clique tree is constructed, it can be used for inference using either of the two message passing algorithms described earlier.

10.5 Summary

In this chapter, we have described a somewhat different perspective on the basic task of exact inference. This approach uses a preconstructed clique tree as a data structure for exact inference. Messages are passed between the cliques in the clique tree, with the end result that the cliques are calibrated — all cliques agree on the same marginal beliefs of any variable they share. We showed two different approaches to message passing in clique trees. The first uses the same operations as variable elimination, using dynamic programming to cache messages in order to avoid repeated computation. The second uses belief propagation messages, which propagate marginal beliefs between cliques in an attempt to make them agree with each other. Both approaches allow calibration of the entire clique tree within two passes over the tree.

It is instructive to compare the standard variable elimination algorithm of chapter 9 and the algorithm obtained by variable elimination in a clique tree. In principle, they are equivalent, in that they both use the same basic operations of multiplying factors and summing out variables. Furthermore, the cliques in the clique tree are basically the factors in variable elimination. Thus, we can use any variable elimination algorithm to find a clique tree, and any clique tree to define an elimination ordering. It follows that the two approaches have basically the same computational complexity.

In practice, however, the two algorithms offer different trade-offs. On one hand, clique trees have several advantages. Most importantly, **through the use of dynamic programming, the clique tree provides answers to multiple cliques using a single computation. Additional layers of dynamic programming allow the same data structure to answer an even broader range of queries, and to dynamically introduce and retract evidence.** Moreover, the clique tree approach executes a nontrivial number of the required operations in advance, including the construction of the basic data structures, the choice of elimination ordering (which is almost determined), and the product of the CPDs assigned to a single clique.

On the other hand, clique trees, as typically implemented, also have disadvantages. First, clique trees are more expensive in terms of space. In a clique tree, we keep all intermediate factors, whereas in variable elimination we can throw them out. If there are c cliques, the cost of the clique tree algorithm can be as much as $2c$ times as expensive. More importantly, **in a clique tree, the structure of the computation is fixed and predetermined. We therefore have less flexibility to take advantage of computational efficiencies that arise because of specific features of the evidence and query.** For example, in the Student network with evidence i^1, the variable elimination algorithm could avoid introducing a dependence between G and S, resulting in substantially smaller factors. In the clique tree algorithm, the clique structure is usually predetermined, precluding these online optimizations. The difference in cost can be

quite dramatic in situations where there is a lot of evidence. This type of situation-specific simplification occurs even more often in networks that exhibit context-specific independence. Finally, in standard implementaitons, the cliques in a clique tree are typically the maximal cliques in a triangulated graph. Furthermore, the operations performed in the clique tree computation are typically implemented in a fairly standard way, where the incoming messages are multiplied with the clique beliefs, and the outgoing message is generated. This approach is not always optimal (see exercise 10.7).

We can modify each of these algorithms to have some of the advantages of the other. For example, we can choose to define a clique tree online, after the evidence is obtained. In this case, the clique tree structure can take advantage of simplifications resulting from the evidence. However, we lose the advantage of precomputing the clique tree offline. As another example, we can store intermediate results in a variable elimination execution, and then do a downward pass to obtain the marginal posteriors of all variables. Here, we gain the advantage of reusing computation, at the cost of additional space. In general, we can view these two algorithms as two examples in a space of variable elimination algorithms. There are many other variants that make somewhat different trade-offs, but, fundamentally, they are performing essentially equivalent computations.

10.6 Relevant Literature

The content of this chapter is closely related to that of chapter 9; hence, many of the citations in section 9.8 are highly relevant to the discussion in this chapter, and the reader is encouraged to explore those as well.

Following the lines of the polytree algorithm, Pearl (1988) proposed a simple approach that clustered nodes so as to produce a polytree; this approach, however, produced very inefficient trees. The sum-product message passing algorithm in clique trees was developed by Shenoy and Shafer (1990); Shafer and Shenoy (1990), who described it in a much broader form that applies to many factored models other than probabilistic graphical models. The sum-product-divide approach was developed in parallel, in a series of papers by Lauritzen and Spiegelhalter (1988) and Jensen, Olesen, and Andersen (1990). This line of work also generated the perspective of message passing operations as performing a reparameterization of the original distribution, an intuition that has been very influential in some of the work of chapter 11. The sum-product-divide algorithm formed the basis for the HUGIN Bayesian network system, described by Andersen, Olesen, Jensen, and Jensen (1989), leading to the common use of the name "HUGIN algorithm" for this method.

HUGIN

Some simple modifications to the clique tree algorithm can greatly improve its efficiency. For example, Kjaerulff (1997) describes a method for improving the in-clique computations by using a *nested clique tree* data structure. Jensen (1995) provides a method for efficiently doing incremental update and retraction of evidence in a clique tree. Park and Darwiche (2004b) provide a derivation of the clique tree algorithm in terms of gradients of the network polynomial (see box 9.A). This approach can also be used as the basis for incremental update and evidence retraction.

nested clique tree

Clique trees and their variants have been extended in many ways, and used for many tasks other than simple probabilistic inference. A complete survey is outside the scope of this book,

but some of these applications and generalizations are mentioned in later chapters.

10.7 Exercises

Exercise 10.1

Prove proposition 10.1.

Exercise 10.2

Prove theorem 10.2.

Exercise 10.3

factor marginalization

Show how to perform efficient index computations for *factor marginalization* along the lines discussed in box 10.A.

Exercise 10.4

Prove theorem 10.5.

Exercise 10.5⋆

Let $\mathcal{T}$ be a clique tree and $\boldsymbol{C}_r$ be a root. Let $\boldsymbol{C}_j$ be a clique in the tree and $\boldsymbol{C}_i$ its upward neighbor. Let β_j be the potential at $\boldsymbol{C}_j$ after the upward pass of CTree-SP-Upward (algorithm 10.1). Show that β_j represents the correct *conditional* probability $\tilde{P}_\Phi(\boldsymbol{C}_j \mid \boldsymbol{S}_{i,j})$. In other words, letting $\boldsymbol{X} = \boldsymbol{C}_j - \boldsymbol{S}_{i,j}$ and $\boldsymbol{S} = \boldsymbol{S}_{i,j}$, we have that:

$$\frac{\beta_j(\boldsymbol{X}, \boldsymbol{S})}{\beta_j(\boldsymbol{S})} = \tilde{P}_\Phi(\boldsymbol{X} \mid \boldsymbol{S}).$$

Exercise 10.6⋆

Assume that we have constructed a clique tree $\mathcal{T}$ for a given Bayesian network graph $\mathcal{G}$, and that each of the cliques in $\mathcal{T}$ contains at most k nodes. Now, the user decides to add a single edge to the Bayesian network, resulting in a network $\mathcal{G}'$. (The edge can be added between any pair of nodes in the network, so long as it maintains acyclicity.) What is the tightest bound you can provide on the maximum clique size in a clique tree $\mathcal{T}'$ for $\mathcal{G}'$? Justify your response by explaining how to construct such a clique tree. (Note: You do not need to provide the optimal clique tree $\mathcal{T}'$. The question asks for the tightest clique tree that you can construct, using only the fact that $\mathcal{T}$ is a clique tree for $\mathcal{G}$.)

Exercise 10.7

The algorithm for performing variable elimination in a clique tree (algorithm 10.1 and algorithm 10.2) specifies a particular approach for sending a variable elimination message: First (in a preprocessing step), the initial clique potentials are generated by multiplying all the factors assigned to a clique. Then, in the message passing operation (SP-Message), the initial potential is multiplied by all of the incoming messages, and the variables not on the sepset are summed out.

Is this the most computationally efficient procedure for executing the message passing step? Either explain why or provide a counterexample.

Exercise 10.8⋆

network polynomial

Consider again the *network polynomial* construction of box 9.A and the algorithm of exercise 9.6 for efficiently computing all the polynomial's derivatives. Show that this algorithm provides an alternative derivation of the up/down clique-tree calibration procedure for sum-product clique trees. In other words, find a correspondence between the computation of the partial derivatives in exercise 9.6 and the message passing operations in the clique tree algorithm.

Exercise 10.9⋆⋆

rule-based clique tree

Use your answer to exercise 10.8 to come up with a *rule-based clique tree* algorithm, based on the rule-based variable elimination procedure of section 9.6.2.1. Your algorithm should compute in a single upward-downward pass all of the marginal probabilities for all variables in the network; its complexity should be twice the cost of the simple rule-based variable elimination of section 9.6.2.1.

Exercise 10.10⋆

Let $\mathcal{T}$ be a calibrated clique tree representing the unnormalized distribution $\tilde{P}_\Phi = \prod_{\phi \in \Phi} \phi$. Let ϕ' be a new factor, and let $\tilde{P}'_\Phi = \tilde{P}_\Phi \cdot \phi'$. Let $\boldsymbol{C}_i$ be some clique such that *Scope*$[\phi] \subseteq \boldsymbol{C}_i$. Show that we can perform an *incremental update* to obtain $\tilde{P}'_\Phi(\boldsymbol{C}_j)$ for any clique $\boldsymbol{C}_j$ by multiplying ϕ' into β_i and then propagating messages from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$ along the path between them.

incremental update

Exercise 10.11⋆

Consider the problem of eliminating extraneous variables from a clique tree. More precisely, given a calibrated clique tree $\mathcal{T}$ over $\mathcal{X}$, we want to generate a (calibrated) clique tree $\mathcal{T}'$ whose scope is some subset $\boldsymbol{Y} \subset \mathcal{X}$. Clearly, we want to make $\mathcal{T}'$ as small as possible (in terms of the size of the resulting cliques). However, we do not want to construct and calibrate a clique tree from scratch; rather, we want to reuse our previous computation.

a. Suppose $\mathcal{T}$ consists of two cliques $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ over variables $\{A, B, C\}$ and $\{C, D\}$, respectively. What is the resulting $\mathcal{T}'$ if $\boldsymbol{Y} = \{B, C\}$?
b. For the clique tree $\mathcal{T}$ defined in part 1, what is $\mathcal{T}'$ if $\boldsymbol{Y} = \{B, D\}$?
c. Now consider an arbitrary clique tree $\mathcal{T}$ over $\mathcal{X}$ and an arbitrary $\boldsymbol{Y} \subseteq \mathcal{X}$. Provide an algorithm to transform a calibrated tree $\mathcal{T}$ into a calibrated tree $\mathcal{T}'$ over $\boldsymbol{Y}$. Your algorithm should *not* resort to manipulating the underlying network or factors; all operations should be performed directly on the clique tree.
d. Give an example where the resulting tree $\mathcal{T}'$ is larger than the original clique tree $\mathcal{T}$.

Exercise 10.12⋆⋆

Let $\mathcal{T}$ be a clique tree over $\mathcal{X}$, defined via a set of initial factors Φ. Let $\boldsymbol{Y} = \boldsymbol{y}$ be some observed assignment to a subset of the variables. Consider a setting where we might be unsure about a particular observation $Y_i = y_i^j$ for $Y_i \in \boldsymbol{Y}$, and that we want to compute the effect on the unnormalized probability of the other possible values y_i^k. More precisely, let $\boldsymbol{Y}_{-i} = \boldsymbol{Y} - \{Y_i\}$ and $\boldsymbol{y}_{-i}$ be the assignment in $\boldsymbol{y}$ to $\boldsymbol{Y}_{-i}$. We want to compute $\tilde{P}_\Phi(y_i^k, \boldsymbol{Y}_{-i} = \boldsymbol{y}_{-i})$ for every Y_i and every y_i^k. Describe a variant of sum-product message passing (algorithm 10.2) that can perform this task without requiring more messages than the standard two-pass calibration. (Hint: Rather than reducing the factors prior to message passing, consider reducing factors during the message passing process.)

Exercise 10.13

Provide a simple method for constructing a clique tree such that a given set of variables $\boldsymbol{Y}$ is guaranteed to be to together in some clique. Your algorithm should use a standard clique-tree construction algorithm as a black box.

Exercise 10.14

Assume that we have a clique tree $\mathcal{T}$ over $\mathcal{X}$ such that, for every pair of nodes $X, Y \in \mathcal{X}$, there exists a clique that contains both X and Y. Prove that $\mathcal{T}$ must contain a single clique that contains all of $\mathcal{X}$.

Exercise 10.15⋆

Consider the task of using a calibrated clique tree $\mathcal{T}$ over Φ to compute all of the pairwise marginals of variables, $\tilde{P}_\Phi(X, Y)$ for all X, Y. Assume that our probabilistic network consists of a chain $X_1 — X_2 — \ldots — X_n$, and that our clique tree has the form $1 — \ldots — n-1$ where *Scope*$[\boldsymbol{C}_i] = \{X_i, X_{i+1}\}$. Also assume that each variable X_i has $|Val(X_i)| = d$.

a. What is the total cost (number of multiplication steps and number of addition steps) of doing variable elimination over this chain-structured clique tree, as described in the algorithm of algorithm 10.4, for all $\binom{n}{2}$ variable pairs?

b. What is the total cost of the algorithm described in section 10.3.3.3? Provide a precise expression, not merely an asymptotic upper bound.

c. Since we are computing marginals for all variable pairs, we may store any computations done for the previous pairs and use them to save time for the remaining pairs. Construct a dynamic programming algorithm for this specific chain structure that reduces the complexity of this task by a factor of d^2 over the algorithm described in section 10.3.3.3? Explain why this approach idea would not work for a general clique tree.

Exercise 10.16

Complete the proof of theorem 10.6 by showing that, if we have a valid clique tree, then a clique elimination step, as described, results in a valid clique tree. In particular, show that it is a tree, that it satisfies the family preservation property, and that it satisfies the running intersection property.

Exercise 10.17⋆

Show that the clique tree constructed using the maximum-weight-spanning tree procedure of section 10.4.2 satisfies the running intersection property. (Hint: Show that this tree structure can also be constructed using the maximum-cardinality algorithm.)

11 *Inference as Optimization*

11.1 Introduction

In the previous chapters we examined exact inference. We have seen that for many networks we can perform exact inference efficiently. As we have seen, the computational and space complexity of the clique tree is exponential in the tree-width of the network. This means that the exact algorithms we examined become infeasible for networks with a large tree-width. In many real-life applications, we encounter such networks. This motivates examination of approximate inference methods that are applicable to networks where exact inference is intractable.

☞ **In this chapter we consider a class of approximate inference methods, where the approximation arises from constructing an approximation to the target distribution P_Φ. This approximation takes a simpler form that allows for inference.** In general, the simpler approximating form exploits a local factorization structure that is similar in nature to the structure exploited by graphical models.

The specific algorithms we consider differ in many details, and yet they share some common conceptual principles. We now review these principles to provide a common framework for the remaining presentation. In each method, we define a target class $\mathcal{Q}$ of "easy" distributions Q and then search for an instance within that class that is the "best" approximation to P_Φ. Queries can then be answered using inference on Q rather than on P_Φ. All of the methods we describe optimize (roughly) the same target function for measuring the similarity between Q and P_Φ.

constrained optimization

This approach reformulates the inference task as one of optimizing an objective function over the class $\mathcal{Q}$. This problem falls into the category of *constrained optimization*. Such problems can be solved using a variety of different methods. Thus, the formulation of inference from this perspective opens the door to the application of a range of techniques developed in the optimization literature. Currently, the technique most often used in the setting of graphical models is one based on the use of *Lagrange multipliers*, which we review in appendix A.5.3. This method produce a set of equations that characterize the optima of the objective. In our setting, this characterization takes the form of a set of fixed-point equations that define each variable in terms of others. A particularly compelling and elegant result is that **the fixed-point equations derived from the constrained energy optimization, for any of the methods we describe, can be viewed as passing messages over a graph object.** Indeed, as we will show, even the standard sum-product algorithm for clique trees (algorithm 10.2) can be rederived from this perspective. Moreover, many other message passing algorithms follow from the same derivation.

☞

Methods in this class fall into three main categories. The first category includes methods that

use clique-tree message passing schemes on structures other than trees. This class of methods, which includes the famous *loopy belief propagation* algorithm, can be understood as optimizing approximate versions of the energy functional. The second category includes methods that use message propagation on clique trees with approximate messages. This class of methods, often known as the *expectation propagation* algorithm, maximize the exact energy functional, but with relaxed consistency constraints on the representation Q. Finally, in the third category there are methods that generalize the *mean field* method originating in statistical physics. These methods use the exact energy functional, but they restrict attention to a class $\mathcal{Q}$ consisting of distributions Q that have a particular simple factorization. This factorization is chosen to be simple enough to ensure that we can perform inference with Q.

More broadly, each of these algorithms can be described from two perspectives: as a procedural description of a message passing algorithm, or as an optimization problem consisting of an objective and a constraint space. Historically, the message passing algorithm generally originated first, sometimes long before the optimization interpretation was understood. However, the optimization perspective provides a much deeper understanding of these methods, and it shows that message passing is only one way of performing the optimization; it also helps point the way toward useful generalizations. In the ensuing discussion, we usually begin the presentation of each class of methods by describing a simple variant of the algorithm, providing a concrete manifestation to ground the concepts. We then present the optimization perspective on the algorithm, allowing a deeper understanding of the algorithm. Finally, we discuss generalizations of the simple algorithm, often ones that are derived directly from the optimization perspective.

11.1.1 Exact Inference Revisited ⋆

Before considering approximate inference methods, we start by casting exact inference as an optimization problem. The concepts we introduce here will serve in the discussion of the following approximate inference methods.

Assume we have a factorized distribution of the form

$$P_\Phi(\mathcal{X}) = \frac{1}{Z} \prod_{\phi \in \Phi} \phi(\boldsymbol{U}_\phi), \tag{11.1}$$

where the factors ϕ in Φ comprise the distribution, and the variables $\boldsymbol{U}_\phi = \mathit{Scope}[\phi] \subseteq \mathcal{X}$ are the scope of each factor. For example, the factors might be CPDs in a Bayesian network, generally restricted by an evidence set, or they might be potentials in a Markov network. We are interested in answering queries about the distribution P_Φ. These include queries about marginal probabilities of variables and queries about the partition function Z. As we discussed, if P_Φ is a Bayesian network with instantiated evidence on some variables, then the partition function Z is the probability of the evidence.

Recall that the end product of belief propagation is a calibrated cluster tree. Also recall that a calibrated set of beliefs for the cluster tree represents a distribution. In exact inference we find a set of calibrated beliefs that represent $P_\Phi(\mathcal{X})$. That is, we find beliefs that match the distribution represented by given set of initial potentials. Thus, we can view exact inference as searching over the set of distributions $\mathcal{Q}$ that are representable by the cluster tree to find a distribution Q^* that matches P_Φ.

Intuitively, we can rephrase this question as searching for a calibrated distribution that is as close as possible to P_Φ. There are many possible ways of measuring the distance between two distributions, such as the Euclidean distance (L_2), or the L_1 distance and the related variational distance (see appendix A.1.3.3). As we will see, our main challenge, however, is our aim to avoid performing inference with the distribution P_Φ; in particular, we cannot effectively compute marginal distributions in P_Φ. Hence, we need methods that allow us to optimize the distance between Q and P_Φ without answering hard queries about P_Φ. A priori, this requirement may seem impossible to satisfy. However, it turns out that there exists a distance measure — the relative entropy (or KL-divergence) — that allows us to exploit the structure of P_Φ without performing reasoning with it.

Recall that the relative entropy between P_1 and P_2 is defined as[1]

$$\boldsymbol{D}(P_1 \| P_2) = \boldsymbol{E}_{P_1}\left[\ln \frac{P_1(\mathcal{X})}{P_2(\mathcal{X})}\right].$$

Also recall that the relative entropy is always nonnegative, and equal to 0 if and only if $P_1 = P_2$. Thus, we can use it as a distance measure, and choose to find an approximation Q to P_Φ that minimizes the relative entropy.

However, as we discussed, the relative entropy is not symmetric — $\boldsymbol{D}(P_1 \| P_2) \neq \boldsymbol{D}(P_2 \| P_1)$. In section 8.5, we discussed the use of relative entropy for projecting a distribution into a restricted class; this projection can aim to minimize either $\boldsymbol{D}(P_\Phi \| Q)$, via the *M-projection*, or $\boldsymbol{D}(Q \| P_\Phi)$, via the *I-projection*. A priori, it might appear that the M-projection is more appropriate, since one of the main information-theoretic justifications for the relative entropy $\boldsymbol{D}(P_\Phi \| Q)$ is the number of bits lost when coding a true message distribution P_Φ using an (approximate) estimate Q. However, as the discussion of section 8.5.2 shows, computing the M-projection Q — $\arg\min_Q \boldsymbol{D}(P_\Phi \| Q)$ — requires that we compute marginals of P_Φ and is therefore equivalent to running inference in P_Φ. Somewhat surprisingly, as we show in the subsequent discussion, this does not apply to I-projection: we can exploit the structure of P_Φ to optimize $\arg\min_Q \boldsymbol{D}(Q \| P_\Phi)$ efficiently, *without* running inference in P_Φ.

M-projection

I-projection

To summarize this discussion, we want to search for a distribution Q that minimizes $\boldsymbol{D}(Q \| P_\Phi)$. To define and analyze this optimization problem formally, we also need to specify the objects we optimize over. Suppose we are given a clique tree structure $\mathcal{T}$ for P_Φ; that is, $\mathcal{T}$ satisfies the running intersection property and the family preservation property. Moreover, suppose we are given a set of beliefs

$$\boldsymbol{Q} = \{\beta_i : i \in \mathcal{V}_\mathcal{T}\} \cup \{\mu_{i,j} : (i\text{–}j) \in \mathcal{E}_\mathcal{T}, \}$$

where $\boldsymbol{C}_i$ denotes clusters in $\mathcal{T}$, β_i denotes beliefs over $\boldsymbol{C}_i$, and $\mu_{i,j}$ denotes beliefs over $\boldsymbol{S}_{i,j}$ of edges in $\mathcal{T}$.

As in definition 10.6, the set of beliefs in $\mathcal{T}$ defines a distribution Q by the formula

$$Q(\mathcal{X}) = \frac{\prod_{i \in \mathcal{V}_\mathcal{T}} \beta_i}{\prod_{(i\text{–}j) \in \mathcal{E}_\mathcal{T}} \mu_{i,j}}. \tag{11.2}$$

1. Note that, until now, we defined the relative entropy and other information-theoretic terms, such as mutual information, using logarithms to base 2. As will become apparent, in the context of the discussion in this chapter, the natural logarithm (base e) is more suitable. This change is a simple rescaling of the relevant information-theoretic quantities and does not change their basic properties.

calibration
marginal consistency

(See section 10.2.3.) Due to the *calibration* requirement, the set of beliefs $\boldsymbol{Q}$ satisfies the *marginal consistency constraints* if, for each $(i\text{-}j) \in \mathcal{E}_{\mathcal{T}}$, the beliefs on $\boldsymbol{S}_{i,j}$ are the marginal of β_i (and β_j). Recall that theorem 10.4 shows that if $\boldsymbol{Q}$ is a set of calibrated beliefs for $\mathcal{T}$ and Q is the distribution defined by equation (11.2), then

$$\begin{aligned} \beta_i[\boldsymbol{c}_i] &= Q(\boldsymbol{c}_i) \\ \mu_{i,j}[\boldsymbol{s}_{i,j}] &= Q(\boldsymbol{s}_{i,j}). \end{aligned}$$

Thus, the beliefs correspond to marginals of the distribution Q defined by equation (11.2).

Thus, we are now searching over a set of distributions Q that are representable by a set of beliefs $\boldsymbol{Q}$ over the cliques and sepsets in a particular clique tree structure $\mathcal{T}$. Note that when deciding on the representation of $\boldsymbol{Q}$ we are actually making two decisions: We are deciding both on the space of distributions that we are considering (all distributions for which $\mathcal{T}$ is an I-map), and on the representation of these distributions (as a set of calibrated clique beliefs). Both of these decisions are significant components in the specification of our optimization problem.

With these definitions in hand, we can now view exact inference as maximizing $-\boldsymbol{D}(Q\|P_\Phi)$ over the space of calibrated sets $\boldsymbol{Q}$.

CTree-Optimize-KL:

Find $\boldsymbol{Q} = \{\beta_i : i \in \mathcal{V}_{\mathcal{T}}\} \cup \{\mu_{i,j} : (i\text{-}j) \in \mathcal{E}_{\mathcal{T}}\}$
maximizing $-\boldsymbol{D}(Q\|P_\Phi)$
subject to

$$\mu_{i,j}[\boldsymbol{s}_{i,j}] = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{c}_i) \quad \forall (i\text{-}j) \in \mathcal{E}_{\mathcal{T}}, \forall \boldsymbol{s}_{i,j} \in Val(\boldsymbol{S}_{i,j})$$

$$\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) = 1 \qquad \forall i \in \mathcal{V}_{\mathcal{T}}.$$

In solving this optimization problem, we conceptually examine different configurations of beliefs that satisfy the marginal consistency constraints, and we select the configuration that maximizes the objective. Such an exhaustive examination, of course, is impossible to perform in practice. However, there are effective solutions to this problem that find the maximum point. We have already seen that, if $\mathcal{T}$ is a proper cluster tree for the set of original potentials Φ, we know that there is a set $\boldsymbol{Q}$ that induces, via equation (11.2), a distribution $Q = P_\Phi$. Because this solution achieves a relative entropy of 0, which is the highest value possible, it is the unique global optimum of this optimization.

Theorem 11.1 *If $\mathcal{T}$ is an I-map of P_Φ, then there is a unique solution to* CTree-Optimize-KL.

This optimum can be found using the exact inference algorithms we developed in chapter 10.

11.1.2 The Energy Functional

The preceding discussion suggests a strategy for constructing approximations of P_Φ. Instead of searching over the space of all calibrated cluster trees, we can search over a space of "simpler" distributions. In this search we will not find a distribution equivalent to P_Φ, yet we might

find one that is reasonably close to P_Φ. Moreover, as part of the design of the target set of distributions, we can ensure that these distributions are ones in which we can perform inference efficiently.

One problem that we will face is that the target of the optimization $\boldsymbol{D}(Q\|P_\Phi)$ is unwieldy for direct optimization. The relative entropy term contains an explicit summation over all possible instantiations of $\mathcal{X}$, an operation that is infeasible in practice. However, since we know the form of $\ln P_\Phi(\xi)$ from equation (11.1), we can exploit its structure to rewrite the relative entropy in a simpler form, as shown in the following theorem.

Theorem 11.2

$\boldsymbol{D}(Q\|P_\Phi) = \ln Z - F[\tilde{P}_\Phi, Q]$
where $F[\tilde{P}_\Phi, Q]$ *is the* energy functional

energy functional

$$F[\tilde{P}_\Phi, Q] = \boldsymbol{E}_Q\Big[\ln \tilde{P}(\mathcal{X})\Big] + \boldsymbol{H}_Q(\mathcal{X}) = \sum_{\phi\in\Phi} \boldsymbol{E}_Q[\ln \phi] + \boldsymbol{H}_Q(\mathcal{X}). \tag{11.3}$$

Proof

$$\boldsymbol{D}(Q\|P_\Phi) = \boldsymbol{E}_Q[\ln Q(\mathcal{X})] - \boldsymbol{E}_Q[\ln P_\Phi(\mathcal{X})]. \tag{11.4}$$

Using the product form of P_Φ, we have that

$$\ln P_\Phi(\mathcal{X}) = \sum_{\phi\in\Phi} \ln \phi(\boldsymbol{U}_\phi) - \ln Z.$$

Moreover, recall that $\boldsymbol{H}_Q(\mathcal{X}) = -\boldsymbol{E}_Q[\ln Q(\mathcal{X})]$. Plugging these into equation (11.4), we get

$$\begin{aligned}\boldsymbol{D}(Q\|P_\Phi) &= -\boldsymbol{H}_Q(\mathcal{X}) - \boldsymbol{E}_Q\left[\sum_{\phi\in\Phi} \ln \phi(\boldsymbol{U}_\phi)\right] + \boldsymbol{E}_Q[\ln Z] \\ &= -F[\tilde{P}_\Phi, Q] + \ln Z.\end{aligned}$$

■

Importantly, the term $\ln Z$ does not depend on Q. Hence, minimizing the relative entropy $\boldsymbol{D}(Q\|P_\Phi)$ is equivalent to maximizing the energy functional $F[\tilde{P}_\Phi, Q]$.

This latter term relates to concepts from statistical physics, and it is the negative of what is referred to in that field as the *(Helmholtz) free energy*. While explaining the physics-based motivation for this term is out of the scope of this book, we continue to use the standard terminology of energy functional.

free energy

The energy functional contains two terms. The first, called the *energy term*, involves expectations of the logarithms of factors in Φ. Here, each factor in Φ appears as a separate term. Thus, if the factors that comprise Φ are small, each expectation deals with relatively few variables. The difficulties in dealing with these expectations depends on the properties of the distribution Q. Assuming that inference is "easy" in Q, we should be able to evaluate such expectations relatively easily. The second term, called the *entropy term*, is the entropy of Q. Again, the choice of Q determines whether we can evaluate this term. However, we will see that, for the choices we make, this term will also be tractable.

energy term

entropy term

11.1.3 Optimizing the Energy Functional

In the remainder of this chapter, **we pose the problem of finding a good approximation Q as one of maximizing the energy functional, or, equivalently, minimizing the relative entropy.** Importantly, the energy functional involves expectations in Q. As we show, by choosing approximations Q that allow for efficient inference, we can both evaluate the energy functional and optimize it effectively.

Moreover, since $\boldsymbol{D}(Q\|P_\Phi) \geq 0$, we have that

$$\ln Z \geq F[\tilde{P}_\Phi, Q]. \tag{11.5}$$

lower bound

That is, the energy functional is a *lower bound* on the logarithm of the partition function Z, for any choice of Q. Why is this fact significant? Recall that, in directed models, the partition function Z is the probability of the evidence. Computing the partition function is often the hardest part of inference. And so, this theorem shows that if we have a good approximation (that is, $\boldsymbol{D}(Q\|P_\Phi)$ is small), then we can get a good lower-bound approximation to Z. The fact that this approximation is a lower bound will play an important role in later chapters on learning.

variational method

In this chapter, we explore inference methods that can be viewed as strategies for optimizing the energy functional. These kinds of methods are often referred to as *variational methods*. The name refers to a general strategy in which we want to solve a problem by introducing new variational parameters that increase the degrees of freedom over which we optimize. Each choice of these parameters gives an approximate answer. We then attempt to optimize the variational parameters to get the best approximation. In our case, the task is to answer queries about P_Φ, and the variational parameters describe the distribution Q. In the methods we consider, we vary these parameters to try to find a good approximation to the target query.

11.2 Exact Inference as Optimization

Before considering approximate inference methods, we illustrate the the use of a variational approach to rederive an exact inference procedure. The concepts we introduce here will serve in discussion of the following approximate inference methods.

As we have already seen, the optimization problem CTree-Optimize-KL has a unique solution. We start by reformulating the optimization problem in terms of the energy functional. As we have seen, maximizing the energy functional is equivalent to minimizing the relative entropy between Q and P_Φ.

Once we restrict attention to calibrated cluster trees, we can further simplify the objective function. More precisely, we can rewrite the energy functional in a factored form as a sum of terms each of which depends directly only on one of the beliefs in $\boldsymbol{Q}$. This form reveals the structure in the distribution, and it is therefore a much better starting point for further analysis. As we will see, this form is also the basis for our approximations in subsequent sections.

Definition 11.1

factored energy functional

Given a cluster tree $\mathcal{T}$ with a set of beliefs $\boldsymbol{Q}$ and an assignment α that maps factors in P_Φ to clusters in $\mathcal{T}$, we define the factored energy functional*:*

$$\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] = \sum_{i \in \mathcal{V}_\mathcal{T}} \boldsymbol{E}_{\boldsymbol{C}_i \sim \beta_i}[\ln \psi_i] + \sum_{i \in \mathcal{V}_\mathcal{T}} \boldsymbol{H}_{\beta_i}(\boldsymbol{C}_i) - \sum_{(i-j) \in \mathcal{E}_\mathcal{T}} \boldsymbol{H}_{\mu_{i,j}}(\boldsymbol{S}_{i,j}), \tag{11.6}$$

where ψ_i is the initial potential assigned to $\boldsymbol{C}_i$:

$$\psi_i = \prod_{\phi, \alpha(\phi)=i} \phi,$$

■

and $\boldsymbol{E}_{\boldsymbol{C}_i \sim \beta_i}[\cdot]$ denotes expectation on the value $\boldsymbol{C}_i$ given the beliefs β_i

Before we prove that the energy functional is equivalent to its factored variant, let us first study its components. The first term is a sum of terms of the form $\boldsymbol{E}_{\boldsymbol{C}_i \sim \beta_i}[\ln \psi_i]$. Recall that ψ_i is a factor (not necessarily a distribution) over the scope $\boldsymbol{C}_i$, that is, a function from $Val(\boldsymbol{C}_i)$ to $\mathbb{R}^+$. Its logarithm is therefore a function from $Val(\boldsymbol{C}_i)$ to $\mathbb{R}$. The beliefs β_i are a distribution over $Val(\boldsymbol{C}_i)$. We can therefore compute the expectation $\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) \ln \psi_i$. The last two terms are entropies of the beliefs associated with the clusters and sepsets in the tree. The important benefit of this reformulation is that all the terms are *local*, in the sense that they refer to a specific belief factor. As we will see, this will make our tasks much simpler.

Proposition 11.1 *If $\boldsymbol{Q}$ is a set of calibrated beliefs for $\mathcal{T}$, and Q is defined by equation (11.2), then*

$$\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] = F[\tilde{P}_\Phi, Q].$$

Proof Note that $\ln \psi_i = \sum_{\phi, \alpha(\phi)=i} \ln \phi$. Moreover, since $\beta_i(\boldsymbol{c}_i) = Q(\boldsymbol{c}_i)$, we conclude that

$$\sum_i \boldsymbol{E}_{\boldsymbol{C}_i \sim \beta_i}[\ln \psi_i] = \sum_\phi \boldsymbol{E}_{\boldsymbol{C}_i \sim Q}[\ln \phi].$$

It remains to show that

$$\boldsymbol{H}_Q(\mathcal{X}) = \sum_{i \in \mathcal{V}_\mathcal{T}} \boldsymbol{H}_{\beta_i}(\boldsymbol{C}_i) - \sum_{(i-j) \in \mathcal{E}_\mathcal{T}} \boldsymbol{H}_{\mu_{i,j}}(\boldsymbol{S}_{i,j}).$$

This equality follows directly from equation (11.2) and theorem 10.4. ■

Using this form of the energy, we can now define the optimization problem. We first need to define the space over which we are optimizing. If Q is factorized according to $\mathcal{T}$, we can represent it by a set of calibrated beliefs. Marginal consistency is a constraint on the beliefs that requires neighboring beliefs to agree on the marginal distribution on their joint subset. It is equivalent to requiring that the beliefs be calibrated. Thus, we pose the following constrained optimization procedure:

CTree-Optimize:

Find $\boldsymbol{Q} = \{\beta_i : i \in \mathcal{V}_\mathcal{T}\} \cup \{\mu_{i,j} : (i\text{-}j) \in \mathcal{E}_\mathcal{T}\}$
maximizing $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$
subject to

$$\mu_{i,j}[\boldsymbol{s}_{i,j}] = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{c}_i) \qquad \forall (i\text{-}j) \in \mathcal{E}_\mathcal{T}, \forall \boldsymbol{s}_{i,j} \in Val(\boldsymbol{S}_{i,j}) \tag{11.7}$$

$$\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) = 1 \qquad \forall i \in \mathcal{V}_\mathcal{T} \tag{11.8}$$

$$\beta_i(\boldsymbol{c}_i) \geq 0 \qquad \forall i \in \mathcal{V}_\mathcal{T}, \boldsymbol{c}_i \in Val(\boldsymbol{C}_i). \tag{11.9}$$

The constraints equation (11.7), equation (11.8), and equation (11.9) ensure that the beliefs in $\boldsymbol{Q}$ are calibrated and represent legal distributions (exercise 11.2).

11.2.1 Fixed-Point Characterization

We can now prove that the *stationary points* of this constrained optimization function — the points at which the gradient is orthogonal to all the constraints — can be characterized by a set of *fixed-point equations*. As we show, these equations turn out to be the update equations in the sum-product belief-propagation procedure (CTree-SP-calibrate in algorithm 10.2). Thus, if we turn these equations into an iterative algorithm, as we will describe, we obtain precisely the belief propagation algorithm in clique trees. We note that for this derivation and other similar ones later in the chapter, we restrict attention to models where all of the potentials are strictly positive (contain no zero entries). Although the results generally hold also for the case of deterministic potentials (zero entries), the proofs are considerably more complex and are outside the scope of this book.

Recall that a stationary point of a function is either a local maximum, a local minimum, or a saddle point. In the optimization problem CTree-Optimize, there is a single global maximum (see theorem 11.1). Although we do not show it here, one can show that it is also the only stationary point (see exercise 11.3), and thus once we find a stationary point, we know that we have found the maximum.

Lagrange multipliers

We want to *characterize* this stationary point by a set of equations that must hold when the choice of beliefs in $\boldsymbol{Q}$ is at the stationary point. Recall that our aim is to maximize the function $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ under the consistency constraints. The method of *Lagrange multipliers*, reviewed in appendix A.5.3, provides us with tools for dealing with constrained optimization. Because the characterization of the stationary point is of central importance to later developments, we examine how to construct such a characterization using the method of Lagrange multipliers.

When using the method of Lagrange multipliers, we start by defining a Lagrangian with a Lagrange multiplier for each of the constraints on the function we want to optimize. In our case, we have the constraints in equation (11.7) and equation (11.8). We note that, in principle, we also need to introduce a Lagrange multiplier for the inequality constraint that ensures that all beliefs are nonnegative. However, as we will see, the assumption that factors are strictly positive implies that the beliefs we construct in the solution to the optimization problem will be nonnegative, and thus we do not need to enforce these constraints actively. We therefore obtain the following Lagrangian:

$$\begin{aligned}\mathcal{J} \;=\; & \tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] \\ & - \sum_{i \in \mathcal{V}_\mathcal{T}} \lambda_i \left(\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) - 1 \right) \\ & - \sum_i \sum_{j \in \mathrm{Nb}_i} \sum_{\boldsymbol{s}_{i,j}} \lambda_{j \to i}[\boldsymbol{s}_{i,j}] \left(\sum_{\boldsymbol{c}_i \sim \boldsymbol{s}_{i,j}} \beta_i(\boldsymbol{c}_i) - \mu_{i,j}[\boldsymbol{s}_{i,j}] \right),\end{aligned}$$

where Nb_i is the neighbors of $\boldsymbol{C}_i$ in the clique tree. We introduce Lagrange multipliers λ_i for each beliefs factor β_i to ensure that it sums to 1. We also introduce, for each pair of neighboring cliques i and j and assignment to their sepset $\boldsymbol{s}_{i,j}$, a Lagrange multiplier $\lambda_{j \to i}[\boldsymbol{s}_{i,j}]$ to ensure

that the marginal distribution of $\boldsymbol{s}_{i,j}$ in β_j is consistent with its value in the sepset beliefs $\mu_{i,j}$. (Note that we also introduce another Lagrange multiplier for the direction $i \rightarrow j$.)

Remember that $\mathcal{J}$ is a function of the clique beliefs $\{\beta_i\}$, the sepset beliefs $\{\mu_{i,j}\}$, and the Lagrange multipliers. To find the maximum of the Lagrangian, we take its partial derivatives with respect to $\beta_i(\boldsymbol{c}_i)$, $\mu_{i,j}[\boldsymbol{s}_{i,j}]$, and the Lagrange multipliers. These last derivatives reconstruct the original constraints. The first two types of derivatives require some work. Differentiating the Lagrangian (see exercise 11.1), we get that

$$\begin{aligned}
\frac{\partial}{\partial \beta_i(\boldsymbol{c}_i)} \mathcal{J} &= \ln \psi_i[\boldsymbol{c}_i] - \ln \beta_i(\boldsymbol{c}_i) - 1 - \lambda_i - \sum_{j \in \mathrm{Nb}_i} \lambda_{j \rightarrow i}[\boldsymbol{s}_{i,j}] \\
\frac{\partial}{\partial \mu_{i,j}[\boldsymbol{s}_{i,j}]} \mathcal{J} &= \ln \mu_{i,j}[\boldsymbol{s}_{i,j}] + 1 + \lambda_{i \rightarrow j}[\boldsymbol{s}_{i,j}] + \lambda_{j \rightarrow i}[\boldsymbol{s}_{i,j}].
\end{aligned}$$

At the stationary point, these derivatives are zero. Equating each derivative to 0, rearranging terms, and exponentiating, we get

$$\begin{aligned}
\beta_i(\boldsymbol{c}_i) &= \exp\{-1 - \lambda_i\} \psi_i[\boldsymbol{c}_i] \prod_{j \in \mathrm{Nb}_i} \exp\{-\lambda_{j \rightarrow i}[\boldsymbol{s}_{i,j}]\} \\
\mu_{i,j}[\boldsymbol{s}_{i,j}] &= \exp\{-1\} \exp\{-\lambda_{i \rightarrow j}[\boldsymbol{s}_{i,j}]\} \exp\{-\lambda_{j \rightarrow i}[\boldsymbol{s}_{i,j}]\}.
\end{aligned}$$

These equations describe beliefs as functions of terms of the form $\exp\{-\lambda_{i \rightarrow j}[\boldsymbol{s}_{i,j}]\}$. In fact, $\mu_{i,j}$ is a product of two such terms (and a constant). This suggests that these terms play the role of a message $\delta_{i \rightarrow j}$. To make this more explicit, we define

$$\delta_{i \rightarrow j}[\boldsymbol{s}_{i,j}] = \exp\left\{-\lambda_{i \rightarrow j}[\boldsymbol{s}_{i,j}] - \frac{1}{2}\right\}.$$

(We add the term $-\frac{1}{2}$ to deal with the additional $\exp\{-1\}$ term, but since this is a multiplicative constant, it is not that crucial.) We can now rewrite the resulting system of equations as

$$\begin{aligned}
\beta_i(\boldsymbol{c}_i) &= \exp\left\{-\lambda_i - 1 + \frac{1}{2}|\mathrm{Nb}_i|\right\} \psi_i(\boldsymbol{c}_i) \prod_{j \in \mathrm{Nb}_i} \delta_{j \rightarrow i}[\boldsymbol{s}_{i,j}] \\
\mu_{i,j}[\boldsymbol{s}_{i,j}] &= \delta_{i \rightarrow j}[\boldsymbol{s}_{i,j}] \delta_{j \rightarrow i}[\boldsymbol{s}_{i,j}].
\end{aligned}$$

Combining these equations with equation (11.7), we now rewrite the message $\delta_{i \rightarrow j}$ as a function of other messages:

$$\begin{aligned}
\delta_{i \rightarrow j}[\boldsymbol{s}_{i,j}] &= \frac{\mu_{i,j}[\boldsymbol{s}_{i,j}]}{\delta_{j \rightarrow i}[\boldsymbol{s}_{i,j}]} \\
&= \frac{\sum_{\boldsymbol{c}_i \sim \boldsymbol{s}_{i,j}} \beta_i(\boldsymbol{c}_i)}{\delta_{j \rightarrow i}[\boldsymbol{s}_{i,j}]} \\
&= \exp\left\{-\lambda_i - 1 + \frac{1}{2}|\mathrm{Nb}_i|\right\} \sum_{\boldsymbol{c}_i \sim \boldsymbol{s}_{i,j}} \psi_i(\boldsymbol{c}_i) \prod_{k \in \mathrm{Nb}_i - \{j\}} \delta_{k \rightarrow i}[\boldsymbol{s}_{i,k}].
\end{aligned}$$

Note that the term $\exp\left\{-\lambda_i - 1 + \frac{1}{2}|\mathrm{Nb}_i|\right\}$ is a constant (since it does not depend on $\boldsymbol{c}_i$), and when we combine these equations with equation (11.8), we can solve for λ_i to ensure that

this constant normalizes the clique beliefs β_i. We note that if the original factors define a distribution that sums to 1, then the solution for λ_i that satisfies equation (11.8) will be one where $\lambda_i = \frac{1}{2}(|\text{Nb}_i| - 1)$, that is, the normalizing constant is 1.

This derivation proves the following result.

Theorem 11.3

A set of beliefs $\boldsymbol{Q}$ is a stationary point of CTree-Optimize *if and only if there exists a set of factors $\{\delta_{i\to j}[\boldsymbol{S}_{i,j}] : (i\text{–}j) \in \mathcal{E}_{\mathcal{T}}\}$ such that*

$$\delta_{i\to j} \propto \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \left(\prod_{k \in \text{Nb}_i - \{j\}} \delta_{k\to i} \right) \tag{11.10}$$

and moreover, we have that

$$\begin{aligned} \beta_i &\propto \psi_i \left(\prod_{j \in \text{Nb}_i} \delta_{j\to i} \right) \\ \mu_{i,j} &= \delta_{j\to i} \cdot \delta_{i\to j}. \end{aligned}$$

fixed-point equations

This theorem characterizes the solution of the optimization problem in terms of *fixed-point equations* that must hold when we find a maximal Q. These fixed-point equations define the relationships that must hold between the different parameters involved in the optimization problem. Most importantly, equation (11.10) defines each message in terms of *other* messages, allowing an easy iterative approach to solving the fixed point equations. These same themes appear in all the approaches we will discuss later in this chapter.

11.2.2 Inference as Optimization

The fixed-point characterization of theorem 11.3 focuses on the relationships that hold at the maximum point (or points). However, they also hint at a way of achieving these relationships. Intuitively, a change in $\boldsymbol{Q}$ that reduces the differences between the left-hand and right-hand side of these equations will get us closer to a maximum point. The most direct way of reducing such discrepancies is to apply the equations as assignments and iteratively apply equations to the current values of the right-hand side to define a new value for the left-hand side.

More precisely, we initialize all of the $\delta_{i\to j}$'s to 1 and then iteratively apply equation (11.10), computing the left-hand side $\delta_{i\to j}$ of each equality in terms of the right-hand side (essentially converting each equality sign to an assignment). Clearly, a single iteration of this process does not usually suffice to make the equalities hold; however, under certain conditions (which hold in a clique tree), we can guarantee that this process converges to a solution satisfying all of the equations in equation (11.10); the other equations are now easy to satisfy.

Each assignment step defined by a fixed-point equation corresponds to a message passing step, where an outgoing message $\delta_{i\to j}$ is defined in terms of incoming messages $\delta_{k\to i}$. The fact that the process requires multiple assignments to converge corresponds to the fact that inference requires multiple message passing steps. In this specific example, a particular order of applying the fixed-point equation reconstructs the sum-product message passing algorithm in cluster trees shown in algorithm 10.2. As we will see, however, when we consider other variants of the optimization problem, the associated fixed-point equations result in new algorithms.

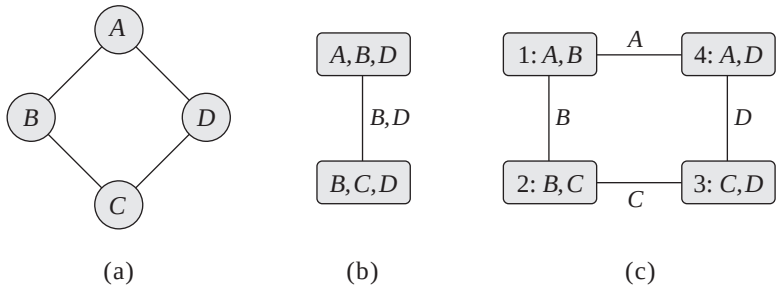


Figure 11.1 An example of a cluster graph. (a) A simple network. (b) A clique tree for the network in (a). (c) A cluster graph for the same network.

11.3 Propagation-Based Approximation

In this section, we consider approximation methods that use exactly the same message propagation as in exact inference. However, these propagation schemes use a general-purpose cluster graph, as in definition 10.1, rather than a clique tree. Since the constraints defining a clique tree were crucial in ensuring exact inference, the message-propagation schemes that use cluster graphs will generally not provide the correct answers.

We begin by defining the general message passing algorithm in a cluster graph. We then show that it can be derived, using the same process as in the previous section, from a set of fixed-point equations induced by the stationary points of an approximate energy functional.

11.3.1 A Simple Example

Consider the simple Markov network of figure 11.1a. Recall that, to perform exact inference within this network, we must first reduce it to a tree, such as the tree of figure 11.1b. Inference in this simple tree involves passing messages over the sepset, which consists of the variables $\{B, D\}$.

Now suppose that, instead, we perform inference as follows. We set up four clusters, which correspond to the four initial potentials: $\boldsymbol{C}_1 = \{A, B\}$, $\boldsymbol{C}_2 = \{B, C\}$, $\boldsymbol{C}_3 = \{C, D\}$, $\boldsymbol{C}_4 = \{A, D\}$. We connect these clusters to each other as shown in the *cluster graph* of figure 11.1c. Note that this cluster graph contains loops (undirected cycles), and is therefore not a tree; such graphs are often called *loopy*. Nevertheless, we can apply the belief-update propagation algorithm CTree-BU-calibrate (algorithm 10.3). Although in our discussion of that algorithm we assumed that the input is a tree, there is nothing in the algorithm itself that relies on that fact. In each step of the algorithm we propagate a message between neighboring clusters. Thus, it is perfectly applicable to a general cluster graph that may not necessarily be a tree.

The clusters in this cluster graph are smaller than those in the clique tree of figure 11.1b; therefore, the message passing steps are less expensive. But what is the result of this procedure? Suppose we propagate messages in the following order $\mu_{1,2}$, $\mu_{2,3}$, $\mu_{3,4}$, and then $\mu_{4,1}$. In the first message, the $\{A, B\}$ cluster passes information to the $\{B, C\}$ cluster through a marginal distribution on B. This information is then propagated to next cluster, and so on. However, in the final message $\mu_{4,1}$, this information reaches the original cluster, but this time as observation

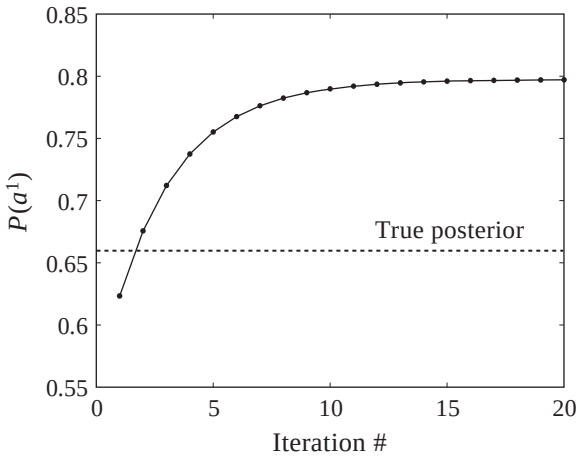


Figure 11.2 An example run of loopy belief propagation in the simple network of figure 11.1a. In this run, all potentials prefer consensus assignments over nonconsensus ones. In each iteration, we perform message passing for all the edges in the cluster graph of figure 11.1b.

about the values of A. As an example, suppose all clusters favor consensus joint assignments; that is, $\beta_1(a^0, b^0)$ and $\beta_1(a^1, b^1)$ are much larger than $\beta_1(a^1, b^0)$ and $\beta_1(a^0, b^1)$, and similarly for the other beliefs. Thus, if the message $\mu_{1,2}$ strengthens the belief that $B = b^1$, then the message $\mu_{2,3}$ will increase the belief in $C = c^1$ and so on. Once we get around the loop, the message $\mu_{4,1}$ will strengthen the support in $A = a^1$. This message will be incorporated into the cluster as though it were independent evidence that did not depend on the initial propagation. Now, if we continue to apply the same sequence of propagations again, we will keep increasing the beliefs in the assignment of $A = a^1$. This behavior is illustrated in figure 11.2. As we can see, in later iterations the procedure overestimates the marginal probability of A. However, the effect of the "feedback" decays until the iterations converge.

This simple experiment already suggests several important issues we need to consider:

convergence

- In the case of cluster trees, we described a sequence of message propagations that calibrate the tree in two passes. Once the tree is calibrated, additional message propagations do not change any of the beliefs. Thus, we can say that the propagation process has *converged*. When we consider our example, it seems clear that **the process may not converge in two passes, since information from one pass will circulate and affect the next round. Indeed, it is far from clear that the propagation of beliefs necessarily converges at all.**

- In the case of cluster trees, we saw that, in a calibrated tree, each cluster of beliefs is the joint marginal of the cluster variables. As our example suggests, for cluster graph propagation, the beliefs on A are not necessarily the marginal probability in P_Φ. Thus, the question is the relationship between the calibrated cluster graph and the actual probability distribution.

Before we address these questions, we present the algorithm in more general terms.

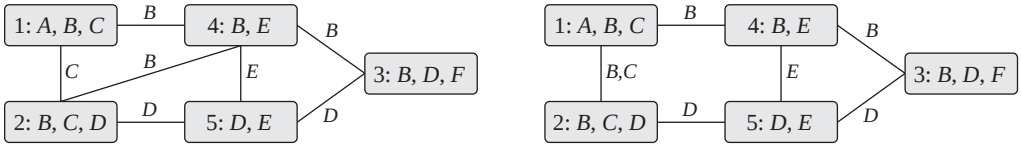


Figure 11.3 Two examples of generalized cluster graph for an MRF with potentials over $\{A, B, C\}$, $\{B, C, D\}$, $\{B, D, F\}$, $\{B, E\}$ and $\{D, E\}$.

Box 11.A — Case Study: Turbocodes and loopy belief propagation. *The idea of propagating messages in loopy graphs was first proposed in the early days of the field, in parallel with the introduction of the first exact inference algorithms. As we discussed in box 9.B, one of the first inference algorithms was Pearl's message passing for singly connected Bayesian networks (polytrees). In his 1988 book, Pearl says:*

> *When loops are present, the network is no longer singly connected and local propagation schemes will invariably run into trouble . . . If we ignore the existence of loops and permit the nodes to continue communicating with each other as if the network were singly connected, messages may circulate indefinitely around the loops and the process may not converge to a stable equilibrium . . . Such oscillations do not normally occur in probabilistic networks . . . which tend to bring all messages to some stable equilibrium as time goes on. However, this asymptotic equilibrium is not coherent, in the sense that it does not represent the posterior probabilities of all nodes of the networks.*

loopy belief propagation

As a consequence of these problems, the idea of loopy belief propagation *was largely abandoned for many years.*

Surprisingly, the revival of loopy belief propagation is due to a seemingly unrelated advance in coding theory. The area of coding addresses the problem of sending messages over a noisy channel, and recovering it from the garbled result. Formally, the coding task can be defined as follows. We wish to send a k-bit message $u_1, \ldots, u_k$. We code *the message using a number of bits $x_1, \ldots, x_n$, which are then sent over the noisy channel, resulting in a set of (possibly corrupted) outputs $y_1, \ldots, y_n$, which can be either discrete or continuous. Different channels introduce noise in different ways: In a simple Gaussian noise model, each bit sent is corrupted independently by the addition of some Gaussian noise; another simple model flips each bit independently with some probability; more complex channel models, where noise is added in a correlated way to consecutive bits, are also used. The* message decoding *task is to recover an estimate $\hat{u_1}, \ldots, \hat{u_k}$ from $y_1, \ldots, y_n$. The* bit error rate *is the probability that a bit is ultimately decoded incorrectly. This error rate depends on the code and decoding algorithm used and on the amount of noise in the channel. The* rate *of a code is k/n — the ratio between the number of bits in the message and the number of bits used to transmit it.*

message decoding

For example, a very simple repetition code takes each bit and transmits it three times, then decodes the bit by majority voting on the three (noisy) copies received. If the channel corrupts each bit with probability p, the bit error rate of this algorithm is $p^3 + 3p^2$, which, for reasonable values

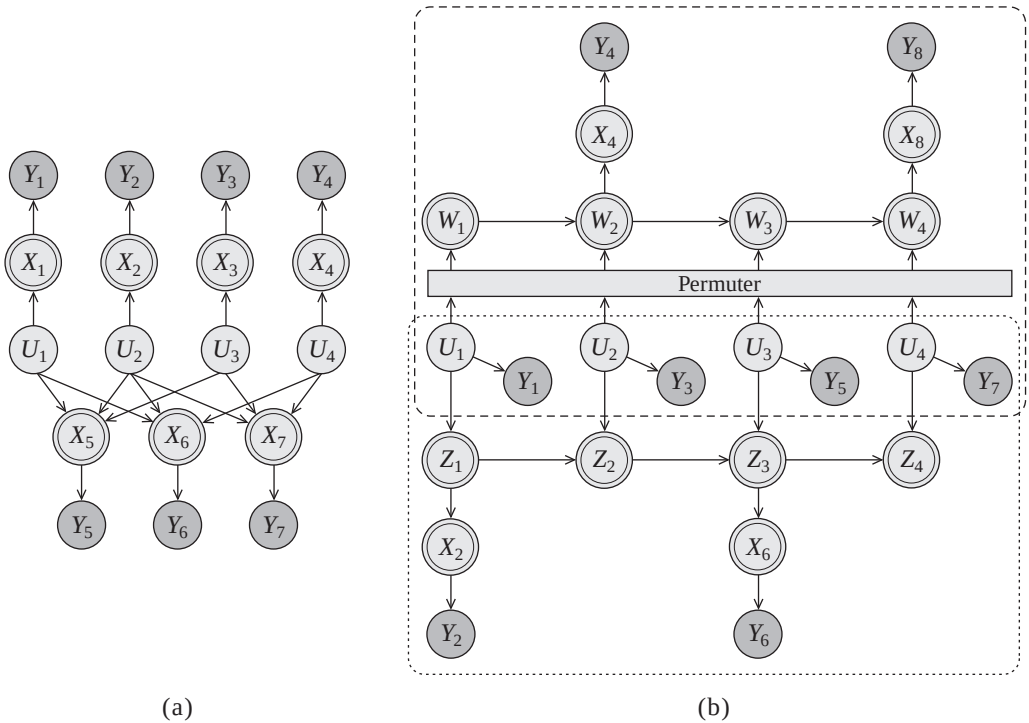


Figure 11.A.1 — Two examples of codes (a) A $k = 4, n = 7$ parity check code, where every four message bits are sent along with three bits that encode parity checks. (b) A $k = 4, n = 8$ turbocode. Here, the $\boldsymbol{X}^a$ bits X_1, X_3, X_5, X_7 are simply the original bits U_1, U_2, U_3, U_4 and are omitted for clarity of the diagram; the $\boldsymbol{X}^b$ bits use a *shift register* — a state bit that changes with each bit of the message, where the ith state bit depends on the $(i-1)$st state bit and on the ith message bit. The code uses two shift registers, one applied to the original message bits and one to a set of permuted message bits (using some predetermined permutations). The sent bits contain both the original message bits and some number of the state bits.

of p, is much lower than p. The rate of this code is $1/3$, because for every message bit, three bits are transmitted. In general, we can get better bit error rates by increasing the redundancy of the code, so we want to compare the bit error rate of different codes that have the same rate. Repetition codes are some of the least efficient codes designed. Figure 11.A.1a shows a simple rate 4/7 parity check code, where every four message bits are sent along with three bits that encode parity checks (exclusive ORs) of different subsets of the four bits.

In 1948, Claude Shannon provided a theoretical analysis of the coding problem (Shannon 1948). For a given rate, Shannon provided an upper bound on the maximum noise level that can be tolerated while still achieving a certain bit error rate, no matter which code is used. Shannon also showed that there exist channel codes that achieve this limit, but his proof was nonconstructive —

he did not present practical encoders and decoders that achieve this limit.

Since Shannon's landmark result, multiple codes were suggested. However, despite a gradual improvement in the quality of the code (bit-error rate for a given noise level), none of the codes even came close to the Shannon limit. The big breakthrough came in the early 1990s, when Berrou et al. (1993) came up with a new scheme that they called a turbocode*, which, empirically, came much closer to achieving the Shannon limit than any other code proposed up to that point. However, their decoding algorithm had no theoretical justification, and, while it seemed to work well in real examples, could be made to diverge or converge to the wrong answer. The second big breakthrough was the subsequent realization that turbocodes were simply performing belief propagation on a Bayesian network representing the probability model for the code and the channel noise.*

turbocode

To understand this, we first observe that message decoding can easily be reformulated as a probabilistic inference task: We have a prior over the message bits $\mathbf{U} = \langle U_1, \ldots, U_k \rangle$, a (usually deterministic) function that defines how a message is converted into a sequence of transmitted bits $X_1, \ldots, X_n$, and another (stochastic) model that defines how the channel randomly corrupts the X_i's to produce Y_i's. The decoding task can then be viewed as finding the most likely joint assignment to $\mathbf{U}$ given the observed message bits $\mathbf{y} = \langle y_1, \ldots, y_n \rangle$, or (alternatively) as finding the posterior $P(U_i \mid \mathbf{y})$ for each bit U_i. The first task is a MAP inference task, and the second task one of computing posterior probabilities. Unfortunately, the probability distribution is of high dimension, and the network structure of the associated graphical model is quite densely connected and with many loops.

The turbocode approach, as first proposed, comprised both a particular coding scheme, and the use of a message passing algorithm to decode it. The coding scheme transmits two sets of bits: one set comprises the original message bits $\mathbf{X}^a = \langle X_1^a, \ldots, X_k^a \rangle = \mathbf{u}$, and the second some set $\mathbf{X}^b = \langle X_1^b, \ldots, X_k^b \rangle$ of transformed bits (like the parity check bits, but more complicated). The received bits then can also be partitioned into the noisy $\mathbf{y}^a, \mathbf{y}^b$. Importantly, the code is designed so that the message can be decoded (albeit with errors) using either $\mathbf{y}^a$ or $\mathbf{y}^b$. The turbocoding algorithm then works as follows: It uses the model of $\mathbf{X}^a$ (trivial in this case) and of the channel noise to compute a posterior probability over $\mathbf{U}$ given $\mathbf{y}^a$. It then uses that posterior $\pi_a(U_1), \ldots, \pi_a(U_k)$ as a prior over $\mathbf{U}$ and computes a new posterior over $\mathbf{U}$, using the model for $\mathbf{X}^b$ and the channel, and $\mathbf{y}^b$ as the evidence, to compute a new posterior $\pi_b(U_1), \ldots, \pi_b(U_k)$. The "new information," which is $\pi_b(U_i)/\pi_a(U_i)$, is then transmitted back to the first decoder, and the process repeats until a stopping criterion is reached. In effect, the turbocoding idea was to use two weak coding schemes, but to "turbocharge" them using a feedback loop. Each decoder is used to decode one subset of received bits, generating a more informed distribution over the message bits to be subsequently updated by the other. The specific method proposed used particular coding scheme for the $\mathbf{X}^b$ bits, illustrated in figure 11.A.1b.

This process looked a lot like black magic, and in the beginning, many people did not even believe that the algorithm worked. However, when the empirical success of these properties was demonstrated conclusively, an attempt was made to understand its theoretical properties. McEliece et al. (1998) subsequently showed that the specific message passing procedure proposed by Berrou et al. is precisely an application of belief propagation (with a particular message passing schedule) to the Bayesian network representing the turbocode (as in figure 11.A.1b).

☞ **This revelation had a tremendous impact on both the coding theory community and the graphical models community. For the former, loopy belief propagation provides a general-purpose algorithm for decoding a large family of codes. By separating the**

algorithmic question of decoding from the question of the code design, it allowed the development of many new coding schemes with improved properties. These codes have come much, much closer to the Shannon limit than any previous codes, and they have revolutionized both the theory and the practice of coding. For the graphical models community, it was the astounding success of loopy belief propagation for this application that led to the resurgence of interest in these approaches, and subsequently to much of the work described in this chapter.

11.3.2 Cluster-Graph Belief Propagation

cluster graph

The basis for our message passing algorithm is the *cluster graph* of definition 10.1, first defined in section 10.1.1. In that section, we required that cluster graphs be trees and that they respect the running intersection property. Those requirements led us to the definition of a clique tree. Here, we remove the first of these two assumptions, allowing inference to be performed on a loopy cluster graph. However, we still wish to require a variant of the running intersection property that is generalized to this case: for any two clusters containing X, there is precisely one path between them over which information about X can be propagated.

Definition 11.2 running intersection property

We say that $\mathcal{U}$ satisfies the running intersection property *if, whenever there is a variable X such that $X \in \boldsymbol{C}_i$ and $X \in \boldsymbol{C}_j$, then there is a single path between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ for which $X \in \boldsymbol{S}_e$ for all edges e in the path.* ■

This generalized running intersection property implies that all edges associated with X form a tree that spans all the clusters that contain X. Thus, intuitively, there is only a single path by which information *that is directly about X* can flow in the graph. Both parts of this assumption are significant. The fact that some path must exist forces information about X to flow between all clusters that contain it, so that, in a calibrated cluster graph, all clusters must agree about the marginal distribution of X. The fact that there is at most one path prevents information about X from cycling endlessly in a loop, making our beliefs more extreme due to "cyclic arguments."

Importantly, however, since the graph is not necessarily a tree, the same pair of clusters might also be connected by other paths. For example, in the cluster graph of figure 11.3a, we see that the edges labeled with B form a subtree that spans all the clusters that contain B. However, there are loops in the graph. For example, there are two paths from $\boldsymbol{C}_3 = \{B, D, F\}$ to $\boldsymbol{C}_2 = \{B, C, D\}$. The first, through $\boldsymbol{C}_4$, propagates information about B, and the second, through $\boldsymbol{C}_5$, propagates information about D. Thus, we can still get circular reasoning, albeit less directly than we would in a graph that did not satisfy the running intersection property; we return to this point in section 11.3.8. Note that while in the case of trees the definition of running intersection implied that $\boldsymbol{S}_{i,j} = \boldsymbol{C}_i \cap \boldsymbol{C}_j$, in a graph this equality is no longer enforced by the running intersection property. For example, cliques $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ in figure 11.3a have B in common, but $\boldsymbol{S}_{1,2} = \{C\}$.

In clique trees, inference is performed by calibrating beliefs. In a cluster graph, we can also associate cluster $\boldsymbol{C}_i$ with *beliefs* β_i. We now say that a cluster graph is *calibrated* if for each

beliefs

calibrated cluster graph

edge $(i$–$j)$, connecting the clusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$, we have that

$$\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j;$$

that is, the two clusters agree on the marginal of variables in $\boldsymbol{S}_{i,j}$. Note that this definition is weaker than cluster tree calibration, since the clusters do not necessarily agree on the joint marginal of all the variables they have in common, but only on those variables in the sepset. However, if a calibrated cluster graph satisfies the running intersection property, then the marginal of a variable X is identical in all the clusters that contain it.

Algorithm 11.1 Calibration using sum-product belief propagation in a cluster graph

```
    Procedure CGraph-SP-Calibrate (
      Φ,    // Set of factors
      U     // Generalized cluster graph Φ
    )
1     Initialize-CGraph
2     while graph is not calibrated
3       Select (i–j) ∈ E_U
4       δ_{i→j}(S_{i,j}) ← SP-Message(i, j)
5     for each clique i
6       β_i ← ψ_i · ∏_{k∈Nb_i} δ_{k→i}
7     return {β_i}

    Procedure Initialize-CGraph (
      U
    )
1     for each cluster C_i
2       β_i ← ∏_{φ : α(φ)=i} φ
3     for each edge (i–j) ∈ E_U
4       δ_{i→j} ← 1
5       δ_{j→i} ← 1
6

    Procedure SP-Message (
      i,    // sending clique
      j     // receiving clique
    )
1     ψ(C_i) ← ψ_i · ∏_{k∈(Nb_i−{j})} δ_{k→i}
2     τ(S_{i,j}) ← ∑_{C_i − S_{i,j}} ψ(C_i)
3     return τ(S_{i,j})
```

How do we calibrate a cluster graph? Because calibration is a local property that relates adjoining clusters, we want to try to ensure that each cluster is sharing information with its

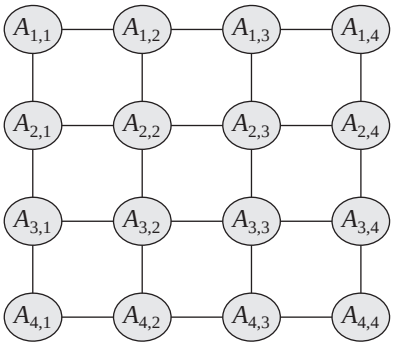


Figure 11.4 An example of a 4×4 two-dimensional grid network

neighbors. From the perspective of a single cluster $\boldsymbol{C}_i$, there is not much difference between a cluster graph and a cluster tree. The cluster is related to each neighboring cluster through an edge that conveys information on variables in the sepset. Thus, we can transmit information by simply having one cluster pass a message to the other.

However, a priori, it is not clear how we can execute a message passing algorithm over a loopy clustergraph. In particular, the sum-product calibration of algorithm 10.2 sends a message only when the sending clique is ready to transmit, that is, when all other incoming messages have been received. In the loopy cluster graph, initially, there is no cluster that has received any incoming messages. Thus, no cluster is ready to transmit, and the algorithm is deadlocked. However, in section 10.3, we showed that the two algorithms are actually equivalent; that is, any sequence of sum-product propagation steps can be emulated by the same sequence of belief-update propagation steps and leads to the same beliefs. In this transformation, we have that $\mu_{i,j} = \delta_{i \to j}\delta_{j \to i}$. Thus, we can construct a "deadlock-free" variant of the sum-product message passing algorithm simply by initializing all messages $\delta_{i \to j} = \mathbf{1}$. This initialization of the sum-product algorithm is equivalent to the standard initialization of the belief update algorithm, in which $\mu_{i,j} = \mathbf{1}$. Importantly, in this variant of the sum-product algorithm, each cluster begins with all of the incoming messages initialized, and therefore it can send any of the outgoing messages at any time, without waiting for any other cluster.

Algorithm 11.1 shows the sum-product message passing algorithm for cluster graphs; other than the fact that the algorithm is applied to graphs rather than trees, the algorithm is identical to CTree-SP-Calibrate. In much the same manner, we can adapt CTree-BU-Calibrate to define a procedure CGraph-BU-Calibrate that operates over cluster graphs using belief-update message passing steps. Both of these algorithms are instances of a general class of algorithms called *cluster-graph belief propagation*, which passes messages over cluster graphs.

cluster-graph belief propagation

Before we continue, we note that cluster-graph belief propagation can be significantly cheaper than performing exact inference. A canonical example of a class of networks that is compactly representable yet hard for inference is the class of grid-structured Markov networks (such as the ones used in image analysis; see box 4.B). In these networks, each variable $A_{i,j}$ corresponds to a point on a two-dimensional grid. Each edge in this network corresponds to a potential between adjacent points on the grid, with $A_{i,j}$ connected to the four nodes $A_{i-1,j}$, $A_{i+1,j}$,

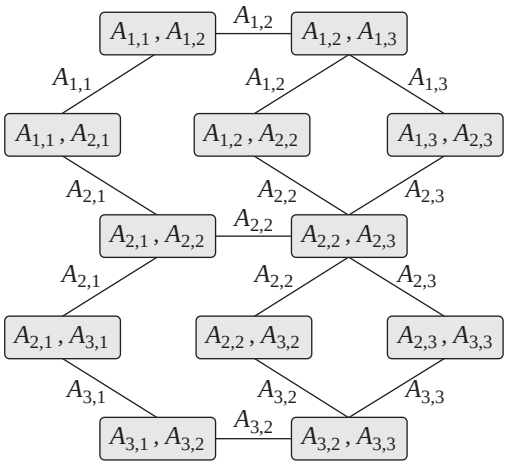


Figure 11.5 An example of generalized cluster graph for a 3×3 grid network

$A_{i,j-1}$, $A_{i,j+1}$ (except for nodes $A_{i,j}$ on the boundary of the grid); see figure 11.4. Such a network has only pairwise potentials, and hence it is very compactly represented. Yet, exact inference requires separating sets, which are as large as cutsets in the grid. Hence, in an $n \times n$ grid, exact computation is exponential in n.

However, we can easily create a generalized cluster graph for grid networks that directly corresponds to the factors in the network. In this cluster graph, each cluster represents beliefs over two neighboring grid variables, and each cluster has a small number of adjoining edges that connect it to other clusters that share one of the two variables. See figure 11.5 for an example for a small 3×3 grid. (Note that there are several ways of constructing such a cluster graph; this figure represents one reasonable choice.) A round of propagations in the generalized cluster graph is linear in the size of the grid (quadratic in n).

11.3.3 Properties of Cluster-Graph Belief Propagation

What can we say about the properties and guarantees provided by cluster-graph belief propagation? We now consider some of the ramifications of the "mechanical" operation of message passing in the graph. Later, when we discuss cluster-graph belief propagation as an optimization procedure, we will revisit this question from a different perspective.

11.3.3.1 Reparameterization

reparameterization

cluster graph invariant

Recall that in section 10.2.3 we showed that belief propagation maintains an invariant property. This allowed us to show that the convergence point represents a *reparameterization* of the original distribution. We can directly extend this property to cluster graphs, resulting in a *cluster graph invariant*.

Theorem 11.4 *Let $\mathcal{U}$ be a generalized cluster graph over a set of factors Φ. Consider the set of beliefs $\{\beta_i\}$ and sepsets $\{\mu_{i,j}\}$ at any iteration of* CGraph-BU-Calibrate*; then*

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

where $\tilde{P}_\Phi(\mathcal{X}) = \prod_{\phi \in \Phi} \phi$ is the unnormalized distribution defined by Φ.

Proof Recall that $\beta_i = \psi_i \prod_{j \in \mathrm{Nb}_i} \delta_{j \to i}$ and that $\mu_{i,j} = \delta_{j \to i} \delta_{i \to j}$. We now have

$$\begin{aligned}
\frac{\prod_{i \in \mathcal{V}_\mathcal{U}} \beta_i[\boldsymbol{C}_i]}{\prod_{(i-j) \in \mathcal{E}_\mathcal{U}} \mu_{i,j}[\boldsymbol{S}_{i,j}]} &= \frac{\prod_{i \in \mathcal{V}_\mathcal{U}} \psi_i[\boldsymbol{C}_i] \prod_{j \in \mathrm{Nb}_i} \delta_{j \to i}[\boldsymbol{S}_{i,j}]}{\prod_{(i-j) \in \mathcal{E}_\mathcal{U}} \delta_{j \to i}[\boldsymbol{S}_{i,j}] \delta_{i \to j}[\boldsymbol{S}_{i,j}]} \\
&= \prod_{i \in \mathcal{V}_\mathcal{U}} \psi_i[\boldsymbol{C}_i] \\
&= \prod_{\phi \in \Phi} \phi(\boldsymbol{U}_\phi) = \tilde{P}_\Phi(\mathcal{X}).
\end{aligned}$$

Note that the second step is based on the fact that each message $\delta_{i \to j}$ appears exactly once in the numerator and the denominator and thus can be canceled. ∎

☞ **This property shows that cluster-graph belief propagation preserves all of the information about the original distribution. In particular, it does not "dilute" the original factors by performing propagation along loops. Hence, we can view the process as trying to represent the original factors anew in a more useful form.**

11.3.3.2 Tree Consistency

Recall that theorem 10.4 implies that, in a calibrated cluster tree, the belief over a cluster is the marginal of the distribution. Thus, in a calibrated cluster tree, we can "read off" the marginals of P_Φ locally from clusters that contain them. More precisely, by normalizing the beliefs factor β_i (so that it sums to 1), we get the marginal distribution over $\boldsymbol{C}_i$. An obvious question is whether a corresponding property holds for cluster-graph belief propagation. Suppose we manage to calibrate a generalized cluster graph and normalize the resulting beliefs; do we have an interpretation for the beliefs in each cluster?

As we saw in our simple example (figure 11.2), the beliefs we compute by BU-message are not necessarily marginals of P_Φ, but rather an approximation. Can we say anything about the quality of this approximation? To characterize the beliefs we get at the end of the process, we can use the cluster tree invariant property applied to subtrees of a cluster graph.

Consider a subtree $\mathcal{T}$ of $\mathcal{U}$; that is, a subset of clusters and edges that together form a tree that satisfies the running intersection property. For example, consider the cluster graph of figure 11.1c. If we remove one of the clusters and its incident edges, we are left with a proper cluster tree. Note that the running intersection property is not necessarily as easy to achieve in general, since removing some edges from the cluster graph may result in a graph that violates the running intersection property relative to a variable, necessitating the removal of additional edges, and so on.

Once we select a tree $\mathcal{T}$, we can think of it as defining a distribution

$$P_{\mathcal{T}}(\mathcal{X}) = \frac{\prod_{i \in \mathcal{V}_{\mathcal{T}}} \beta_i(\boldsymbol{C}_i)}{\prod_{(i-j) \in \mathcal{E}_{\mathcal{T}}} \mu_{i,j}[\boldsymbol{S}_{i,j}]}.$$

If the cluster graph is calibrated, then by definition so is $\mathcal{T}$. And so, because $\mathcal{T}$ is a tree that satisfies the running intersection property, we can apply theorem 10.4, and we conclude that

$$\beta_i(\boldsymbol{C}_i) = P_{\mathcal{T}}(\boldsymbol{C}_i). \tag{11.11}$$

tree consistency

That is, the beliefs over $\boldsymbol{C}_i$ in the tree are the marginal of $P_{\mathcal{T}}$, a property called *tree consistency*.

As a concrete example, consider the cluster graph of figure 11.1c. Removing the cluster $\boldsymbol{C}_4 = \{A, D\}$, we are left with a proper cluster tree $\mathcal{T}$. The preceding argument implies that once we have calibrated the cluster graph, we have $\beta_1(A, B) = P_{\mathcal{T}}(A, B)$. This result suggests that $\beta_1(A, B) \neq P_\Phi(A, B)$; to show this formally, contrast equation (11.11) with theorem 11.4. We see that the tree distribution involves some of the terms that define the joint distribution. Thus, we can conclude that

$$P_{\mathcal{T}}(A, B, C, D) = P_\Phi(A, B, C, D) \frac{\mu_{3,4}[D]\mu_{1,4}[A]}{\beta_4(A, D)}.$$

We see that unless $\beta_4(A, D) = \mu_{3,4}[D]\mu_{1,4}[A]$, $P_{\mathcal{T}}$ will be different from P_Φ. This conclusion suggests that, in this example, the beliefs $\beta_1(A, B)$ in the calibrated cluster graph are not the marginal $P_\Phi(A, B)$.

Clearly, we can apply the same type of reasoning using other subtrees of $\mathcal{U}$. And so we reach the surprising conclusion that equation (11.11) must hold with respect to every cluster tree embedded in $\mathcal{U}$. In our example, we can see that by removing a single cluster, we can construct three different trees that contain $\boldsymbol{C}_1$. The same beliefs $\beta_1(A, B)$ are the marginal of the three distributions defined by each of these trees. While these three distributions agree on the joint marginal of A and B, they can differ on the joint marginal distributions of other pairs of variables.

cluster graph residual

Moreover, these subtrees allow us to get insight about the quality of the marginal distributions we read from the calibrated cluster graph. Consider our example again: we can use the *residual* term $\frac{\mu_{3,4}[D]\mu_{1,4}[A]}{\beta_4(A,D)}$ to analyze the error in the marginal distribution. In this simple example, this analysis is fairly straightforward (see exercise 11.4).

In other cases, the analysis can be more complex. For example, suppose we want to find a subtree in the cluster graph for a grid (e.g., figure 11.5). To construct a tree, we must remove a nontrivial number of clusters. More precisely, because each cluster corresponds to an edge in the grid, a cluster tree corresponds to a subtree of the grid. For an $n \times n$ grid, such a tree will have at most $n^2 - 1$ edges of the $2n(n-1)$ edges in the grid. Thus, each cluster tree contains about half of the clusters in the original cluster graph. In such a situation the residual term is more complex, and we cannot necessarily evaluate it.

11.3.4 Analyzing Convergence ⋆

A key question regarding the belief propagation algorithm is whether and when it converges. Indeed, there are many networks for which belief propagation does not converge; see box 11.C.

Although we cannot hope for convergence in all cases, it is important to understand when this algorithm does converge. We know that if the cluster graph is a tree then the algorithm will converge. Can we find other classes of cluster graphs for which we can prove convergence?

synchronous BP

One method of analyzing convergence is based on the following important perspective on belief propagation. This analysis is easier to perform on a variant of BP called *synchronous BP* that performs all of the message updates simultaneously. Consider the update step that takes all of the messages $\boldsymbol{\delta}^t$ at a particular iteration t and produces a new set of messages $\boldsymbol{\delta}^{t+1}$ for the next step. Letting Δ be the space of all possible messages in the cluster graph, we can view the belief-propagation update operator as a function $G_{BP} : \Delta \mapsto \Delta$. Consider the standard sum-product message update:

$$\delta'_{i \to j} \propto \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in (\mathrm{Nb}_i - \{j\})} \delta_{k \to i},$$

BP operator

where we normalize each message to sum to 1; this renormalization step is essential to avoid a degenerate convergence to the $\mathbf{0}$ message. We can now define the *BP operator* as the function that simultaneously takes one set of messages and computes a new one:

$$G_{BP}(\{\delta_{i \to j}\}) = \{\delta'_{i \to j}\}.$$

The question of convergence of the algorithm now reduces to one of asking whether repeated applications of the operator G_{BP} are guaranteed to converge.

One interesting, albeit strong, condition that guarantees convergence is the *contraction property*:

Definition 11.3 contraction

For a number $\alpha \in [0, 1)$, an operator G over a metric space $(\Delta, \mathbb{D}(;))$ is an α-contraction *relative to the distance function $\mathbb{D}(;)$ if, for any $\boldsymbol{\delta}, \boldsymbol{\delta}' \in \Delta$, we have that:*

$$\mathbb{D}(G(\boldsymbol{\delta}); G(\boldsymbol{\delta}')) \leq \alpha \mathbb{D}(\boldsymbol{\delta}; \boldsymbol{\delta}'). \tag{11.12}$$

■

In other words, an operator is a contraction if its application to two points in the space is guaranteed to decrease the distance between them by at least some constant factor $\alpha < 1$.

A basic result in analysis shows that, under fairly weak conditions, if an operator G is a contraction, we have that repeated applications of G are guaranteed to converge to a unique fixed point:

Proposition 11.2 fixed-point

Let G be an α-contraction of a complete metric space $(\Delta, \mathbb{D}(;))$. Then there is a unique fixed-point *$\boldsymbol{\delta}^*$ for which $G(\boldsymbol{\delta}^*) = \boldsymbol{\delta}^*$. Moreover, for any $\boldsymbol{\delta}$, we have that*

$$\lim_{n \to \infty} G^n(\boldsymbol{\delta}) = \boldsymbol{\delta}^*.$$

The proof is left as an exercise (exercise 11.5).

Indeed, the *contraction rate* α can be used to provide bounds on the rate of convergence of the algorithm to its unique fixed point: To reach a point that is guaranteed to be within ϵ of $\boldsymbol{\delta}^*$, it suffices to apply G the following number of times:

$$\log_\alpha \frac{\epsilon}{\mathrm{diameter}(\Delta)},$$

where $\text{diameter}(\Delta) = \max_{\boldsymbol{\delta}, \boldsymbol{\delta}' \in \Delta} \boldsymbol{D}(\boldsymbol{\delta}; \boldsymbol{\delta}')$.

Applying this analysis to the operator G induced by the belief-propagation message update is far from trivial. This operator is complex and nonlinear, because it involves both multiplying messages and a renormalization step. A review of these analyses is outside the scope of this book. At a high level, these results show that if the factors in the network are fairly "smooth," one can guarantee that the synchronous BP operator is a contraction and hence converges to a unique fixed point. We describe one of the simplest of these results, in order to give a flavor for this type of analysis.

This analysis applies to synchronous loopy belief propagation over a pairwise Markov network with two-valued random variables $X_i \in \{-1, +1\}$. Specifically, we assume that the network model is parameterized as follows:

$$P(x_1, \ldots, x_n) = \frac{1}{Z} \exp\left(\sum_{(i,j)} \epsilon_{i,j}(x_i, x_j) + \sum_i \epsilon_i(x_i), \right),$$

where we assume for simplicity of notation that $\epsilon_{i,j} = 0$ when X_i and X_j are not neighbors in the network.

hyperbolic tangent

We begin by introducing some notation. The *hyperbolic tangent* function is defined as:

$$\tanh(w) = \frac{e^w - e^{-w}}{e^w + e^{-w}} = \frac{e^{2w} - 1}{e^{2w} + 1}.$$

The hyperbolic tangent has a shape very similar to the sigmoid function of figure 5.11a. The following condition can be shown to suffice for G_{BP} to be a contraction, and hence for the convergence of belief propagation to a unique fixed point:

$$\max_i \max_{j \in \text{Nb}_i} \sum_{k \in \text{Nb}_i - \{j\}} \tanh |\epsilon_{k,i}| < 1. \tag{11.13}$$

Intuitively, this expression measures the total extent to which i's neighbors other than j can influence the message from i to j. The larger the magnitude of the parameters in the network, the larger this sum.

The analysis of the more general case is significantly more complex but shares the same intuitions. At a very high level, if we can place strong bounds on the skew of the parameters in a factor:

$$\max_{\boldsymbol{x}, \boldsymbol{x}'} \phi(\boldsymbol{x}) / \phi(\boldsymbol{x}'),$$

we can guarantee convergence of belief propagation. Intuitively, the lower the skew of the factors in our network, the more each message update "smoothes out" differences between entries in the messages, and therefore also makes different messages more similar to each other.

While the conditions that underlie these theorems are usually too stringent to hold in practice, this analysis does provide useful insight. First, it suggests that **networks with potentials that are closer to deterministic are more likely to have problems with convergence, an observation that certainly holds in practice.** Second, although global contraction throughout the space is a very strong assumption, a contraction property in a region of the space may be plausible, guaranteeing convergence of the algorithm if it winds up (or is initialized) in this region. These results and their ramifications are only now being explored.

11.3.5 Constructing Cluster Graphs

So far, we have taken the cluster graph to be given. However, the choice of cluster graph is generally far from obvious, and it can make a significant difference to the algorithm. Recall that, even in exact inference, more than one clique tree can be used to perform inference for a given distribution. However, while these different trees can vary in their computational cost, they all give rise to the same answers. **In the case of cluster graph approximations, different graphs can lead to very different answers. Thus, when selecting a cluster graph, we have to consider trade-offs between cost and accuracy, since cluster graphs that allow fast propagation might result in a poor approximation.**

It is important to keep in mind that the structure of the cluster graph determines the propagation steps the algorithm can perform, and thus dictate what type of information is passed during the propagations. These choices directly influence the quality of the results.

Example 11.1

Consider, for example, the cluster graphs $\mathcal{U}_1$ and $\mathcal{U}_2$ of figure 11.3a and figure 11.3b. Both are fairly similar, yet in $\mathcal{U}_2$ the edge between $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ involves the marginal distribution over B and C. On the other hand, in $\mathcal{U}_1$, we propagate the marginal only over C. Intuitively, we expect inference in $\mathcal{U}_2$ to better capture the dependencies between B and C. For example, assume that the potential of $\boldsymbol{C}_1$ introduces strong correlations between B and C (say $B = C$). In $\mathcal{U}_2$, this correlation is conveyed to $\boldsymbol{C}_2$ directly. In $\mathcal{U}_1$, the marginal on C is conveyed on the edge $(1\text{–}2)$, while the marginal on B is conveyed through $\boldsymbol{C}_4$. In this case, the strong dependency between the two variables is lost. In particular, if the marginal on C is diffuse (close to uniform), then the message $\boldsymbol{C}_1$ sends to $\boldsymbol{C}_4$ will also have a uniform distribution on B, and from $\boldsymbol{C}_2$'s perspective the messages on B and C will appear as two independent variables. ■

On the other hand, if we introduce many messages between clusters or increase the scope of these messages, we run the risk of constructing a tree that violates the running intersection property. And so, we have to worry about methods that ensure that the resulting structure is a proper cluster graph. We now consider several approaches for constructing cluster graphs.

11.3.5.1 Pairwise Markov Networks

pairwise Markov networks

We start with the class of *pairwise Markov networks*. In these networks, we have a univariate potential $\phi_i[X_i]$ over each variable X_i, and in addition a pairwise potential $\phi_{(i,j)}[X_i, X_j]$ over some pairs of variables. These pairwise potentials correspond to edges in the Markov network. Many problems are naturally formulated as pairwise Markov networks, including the grid networks we discussed earlier and Boltzmann distributions (see box 4.C). Indeed, if we are willing to transform our variables, any distribution can be reformulated as a pairwise Markov network (see exercise 11.10).

One straightforward transformation of such a network into a cluster graph is as follows: For each potential, we introduce a corresponding cluster, and put edges between the clusters that have overlapping scope. In other words, there is an edge between the cluster $\boldsymbol{C}_{(i,j)}$ that corresponds to the edge X_i—X_j and the clusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ that correspond to the univariate factors over X_i and X_j. Figure 11.6 illustrates this construction in the case of a 3 by 3 grid network.

Because there is a direct correspondence between the clusters in the cluster graphs and vari-

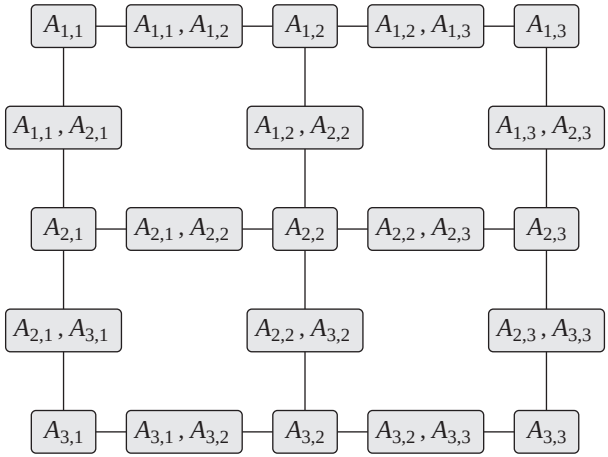


Figure 11.6 A generalized cluster graph for the 3×3 grid when viewed as pairwise MRF

ables or edges in the original Markov network, it is often convenient to think of the propagation steps as operations on the original network. Moreover, since each pairwise cluster has only two neighbors, we consider two propagation steps along the path $\boldsymbol{C}_i — \boldsymbol{C}_{(i,j)} — \boldsymbol{C}_j$ as propagating information between X_i and X_j. (See exercise 11.9.) Indeed, early versions of cluster-graph belief propagation were stated in these terms. This algorithm is known as *loopy belief propagation*, since it uses propagation steps used by algorithms for Markov trees, except that it was applied to networks with loops.

loopy belief propagation

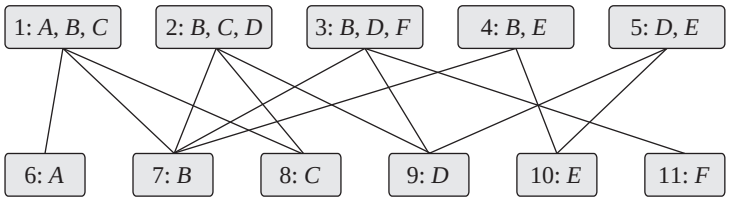
11.3.5.2 Bethe Cluster Graph

A natural question is how we can extend this idea to networks that are more complex than pairwise Markov networks. Once we have larger potentials, they may overlap in ways that result in complex interactions among them.

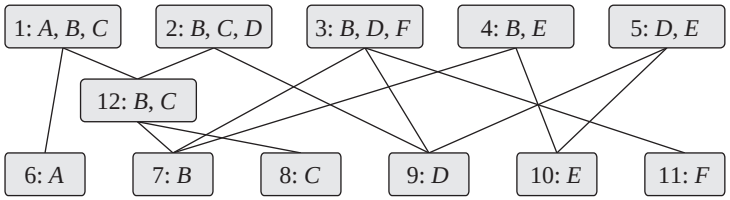
Bethe cluster graph

One simple construction, called the *Bethe cluster graph*, uses a bipartite graph. The first layer consists of "large" clusters, with one cluster for each factor ϕ in Φ, whose scope is $Scope[\phi]$. These clusters ensure that we satisfy the family-preservation property. The second layer consists of "small" univariate clusters, one for each random variable. Finally, we place an edge between each univariate cluster X on the second layer and each cluster in the first layer that includes X; the scope of this edge is X itself. For a concrete example, see figure 11.7a.

We can easily verify that this cluster graph is a proper one. First, by construction, it satisfies the family preservation property. Second, the edges that mention a variable X form a star-shaped subgraph with edges from the univariate cluster for X to all the large clusters that contain X. It is also easy to check that, if we apply this procedure to a pairwise Markov network, it results in the "natural" cluster graph for the pairwise network that we discussed. The construction of this cluster graph is simple and can easily be automated.



(a) $\mathcal{U}_3$



(b) $\mathcal{U}_4$

Figure 11.7 Examples of generalized cluster graphs for network with potentials over $\{A, B, C\}$, $\{B, C, D\}$, $\{B, D, F\}$, $\{B, E\}$ and $\{D, E\}$. For visual clarity, sepsets have been omitted — the sepset between any pair of clusters is the intersection of their scopes. (a) Bethe factorization. (b) Capturing interactions between $\{A, B, C\}$ and $\{B, C, D\}$.

11.3.5.3 Beyond Marginal Probabilities

The main limitation of using the Bethe cluster graph is that information between different clusters in the top level is passed through univariate marginal distributions. Thus, interactions between variables are lost during propagations. Consider the example of figure 11.7a. Suppose that $\boldsymbol{C}_1$ creates a strong dependency between B and C. These two variables are shared with $\boldsymbol{C}_2$. However, the messages between two clusters are mediated through the univariate factors. And thus, interactions introduced by one cluster are not directly propagated to the other.

One possible solution is to merge some of the large clusters. For example, if we want to capture the interactions between $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ in figure 11.7a, we can replace both of them by a cluster with the score A, B, C, D. This new cluster will allow us to capture the interactions between the factors involved in these two clusters. This modification, however, comes at a price, since the cost of manipulating a cluster grows exponentially with this scope. Moreover, this approach seems excessive in this case, since we can summarize these interactions simply using a distribution over B and C. This intuition suggests the construction of figure 11.7b. Note that this cluster graph is equivalent to figure 11.3b; see exercise 11.6.

Can we generalize this construction? A reasonable goal might be to capture all pairwise interactions. We can try to use a construction similar to the Bethe approximation, but introducing an intermediate level that includes pairwise clusters. In the same manner as we introduced $\boldsymbol{C}_{12}$ in figure 11.7b, we can introduce other pairs that are shared by more than two clusters. As a concrete example, consider the factors $\boldsymbol{C}_1 = \{A, B, C\}$, $\boldsymbol{C}_2 = \{B, C, D\}$, and $\boldsymbol{C}_3 = \{A, C, D\}$. The relevant pairwise factors that capture interactions among these clusters

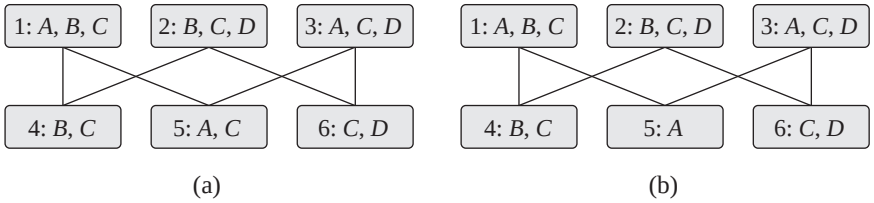


Figure 11.8 Examples of generalized cluster graph for network with potentials over $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, C, D\}$. For visual clarity, sepsets have been omitted — the sepset between any pair of clusters is the intersection of their scopes. (a) A Bethe-like factorization with pairwise marginals that leads to an illegal cluster graph. (b) One possible way to make this graph legal.

are $\{B, C\} = \boldsymbol{C}_1 \cap \boldsymbol{C}_2$, $\{C, D\} = \boldsymbol{C}_2 \cap \boldsymbol{C}_3$, and $\{A, C\} = \boldsymbol{C}_1 \cap \boldsymbol{C}_3$. The resulting cluster graph appears in figure 11.8a. Unfortunately, a quick check shows that this cluster graph does *not* satisfy the running intersection property — all the edges in this graph are labeled by C, and together they form a loop. As a result, information concerning C can propagate indefinitely around the loop, "overcounting" the effect of C in the result.

How do we avoid this problem? In this specific example, we can consider a weaker approximation by removing C from one of the intersection sets. For example, if we remove C from $\boldsymbol{C}_5$, we get the cluster graph of figure 11.8b. This cluster graph satisfies the running intersection property. An alternative approach tries to "compensate" somehow for the violation of the running intersection property using a more complex message passing algorithm; see section 11.3.7.3.

---

belief propagation nonconvergence

Box 11.B — Skill: Making loopy belief propagation work in practice. *One of the main problems with loopy belief propagation is* nonconvergence. *This problem is particularly serious when we build systems that use inference as a subroutine within other tasks, for example, as the inner loop of a learning algorithm (see, for example, section 20.5.1). Several approaches have been used for addressing this nonconvergence issue. Some are fairly simple heuristics. Others are more sophisticated, and typically are based on the characterization of cluster-graph belief propagation as optimizing the approximate free-energy functional.*

A first observation is that, often, nonconvergence is a local problem. In many practical cases, most of the beliefs in the network do converge, and only a small portion of the network remains problematic. In such cases, it is often quite reasonable simply to stop the algorithm at some point (for example, when some predetermined amount of time has elapsed) and use the beliefs at that point, or a running average of the beliefs over some time window. This heuristic is particularly reasonable when we are not interested in individual beliefs, but rather in some aggregate over the entire network, for example, in a learning setting.

A second observation is that nonconvergence is often due to oscillations in the beliefs (see section 11.3.1). This observation suggests that we dampen the oscillations by reducing the difference between two subsequent updates. Consider the belief-propagation update rule in SP-Message(i, j)*:*

$$\delta_{i \to j} \leftarrow \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \prod_{k \neq j} \delta_{k \to i}.$$

damping

We can replace this line by a damped *(or* smoothed*) version that averages the update $\delta_{i\to j}$ with the previous message between the two cliques:*

$$\delta_{i\to j} \leftarrow \lambda \left(\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \prod_{k \neq j} \delta_{k\to i} \right) + (1-\lambda)\delta_{i\to j}^{\text{old}}, \tag{11.14}$$

where λ is the damping weight and $\delta_{i\to j}^{\text{old}}$ is the previous value of the message. When $\lambda = 1$, this update is equivalent to standard belief propagation. For $0 < \lambda < 1$, the update is partial and although it shifts β_j toward agreement with β_i, it leaves some momentum for the old value of the belief, a dampening effect that in turn reduces the fluctuations in the beliefs. It turns out that this damped update rule is "equivalent" to the original update rule, in that a set of beliefs is a convergence point of the damped update if and only if it is a convergence point of standard updates (see exercise 11.13). Moreover, one can show that, if run from a point close enough to a stable convergence point *of the algorithm, with a sufficiently small λ, this damped update rule is guaranteed to converge. Of course, this guarantee is not very useful in practice, but there are indeed many cases where the damped update rule is convergent, whereas the original update rule oscillates indefinitely.*

stable convergence point

message scheduling

A broader-spectrum heuristic, which plays an important role not only in ensuring convergence but also in speeding it up considerably, is intelligent message scheduling*. It is tempting to implement BP message passing as a synchronous algorithm, where all messages are updated at once. It turns out that, in most cases, this schedule is far from optimal, both in terms of reaching convergence, and in the number of messages required for convergence. The latter problem is easy to understand: In a cluster graph with m edges, and diameter d, synchronous message passing requires $m(d-1)$ messages to pass information from one side of the graph to the other. By contrast, asynchronous message passing, appropriately scheduled, can pass information between two clusters at opposite ends of the graph using $d-1$ messages. Moreover, the fact that, in synchronous message passing, each cluster uses messages from its neighbors that are based on their previous beliefs appears to increase the chances of oscillatory behavior and nonconvergence in general.*

asynchronous BP

tree reparameterization

In practice, an *asynchronous* message passing schedule works significantly better than the synchronous approach. Moreover, even greater improvements can be obtained by scheduling messages in a guided way. *One approach, called* tree reparameterization (TRP)*, selects a set of trees, each of which spans a large number of the clusters, and whose union covers all of the edges in the network. The TRP algorithm then iteratively selects a tree and does an upward-downward calibration of the tree, keeping all other messages fixed. Of course, calibrating this tree has the effect of "uncalibrating" other trees, and so this process repeats. This approach has the advantage of passing information more globally within the graph. It therefore converges more often, and more quickly, than other asynchronous schedules, particularly if the trees are selected using a careful design that accounts for the properties of the problem.*

An even more flexible approach attempts to detect dynamically in which parts of the network messages would be most useful. Specifically, as we observed, often some parts of the network converge fairly quickly, whereas others require more messages. We can schedule messages in a way that accounts for their potential usefulness; for example, we can pass a message between clusters where the beliefs disagree most strongly on the sepset. This approach, called residual belief propagation *is convenient, since it is fully general and does not require a deep understanding of the properties of the network. It also works well across a range of different real-world networks.*

residual belief propagation

An alternative general-purpose approach to avoiding nonconvergence is to directly optimize the energy functional. Here, several methods have been proposed. The simplest is to use standard optimization methods such as gradient ascent to optimize $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ (see appendix A.5.2 and exercise 11.12). Other methods are more specialized to the form of the energy functional, and they often turn out to be more efficient (see section 11.7). Although these methods do improve convergence, they are somewhat complex to implement, and have not (at this time) been used extensively in practice.

It turns out that many of the parameter settings encountered during a learning algorithm are problematic, and cause cluster-graph belief propagation to diverge. Intuitively, in many real-world problems, "appropriate" parameters encode strong constraints that tend to drive the algorithm toward well-behaved regions of the space. However, the parameters encountered during an iterative learning procedure have no such properties, and often allow the algorithm to end up in difficult regions. One approach is to train some parameters of the model separately, using a simpler network. We then use these parameters as our starting point in the general learning procedure. The use of "reasonable" parameters in the model can stabilize BP, allowing it to converge within the context of the general learning algorithm.

local maxima

A final problem with cluster-graph belief propagation is the fact that the energy functional objective is multimodal, and so there are many local maxima *to which a cluster-graph belief propagation algorithm might converge (if it converges). One can, of course, apply any of the standard approaches for addressing optimization of multimodal functions, such as initializing the algorithm heuristically, or using multiple restarts with different initializations. In the setting of BP, initialization must be done with care, so as not to lose the connection to the correct underlying distribution P_Φ, as reflected by the invariant of theorem 11.4. In sum-product belief propagation, we can simply initialize the messages to something other than 1. In belief update propagation, care must be taken to initialize messages and beliefs in a coordinate way, to preserve P_Φ.*

Box 11.C — Case Study: BP in practice. *To convey the behavior of belief propagation in practice, we demonstrate its performance on an 11×11 (121 binary variables) Ising grid (see box 4.C). The potentials of the network were randomly sampled as follows: Each univariate potential was sampled uniformly in the interval $[0, 1]$; for each pair of variables X_i, Z_j, $w_{i,j}$ is sampled uniformly in the range $[-C, C]$ (recall that in an Ising model, we define the negative log potential $\epsilon_{i,j}(x_i, x_j) = -w_{i,j}x_i x_j$). This sampling process creates an energy function where some potentials are attractive ($w_{i,j} > 0$) and some are repulsive ($w_{i,j} < 0$), resulting in a nontrivial inference problem. The magnitude of C (11 in this example) controls the magnitude of the energy forces and higher values correspond, on average, to more challenging inference problems.*

Figure 11.C.1 illustrates the convergence behavior on this problem. Panel (a) shows the percentage of messages converged as a function of time for three variants of the belief propagation algorithm: synchronous BP with damping (dashed line), where only a small fraction of the messages ever converge; asynchronous BP with damping (smoothing) that converges (solid line); asynchronous BP with no damping (dash-dot line) that does not fully converge. The benefit of using asynchronous propagation over synchronous updating is obvious. At first, it appears as if smoothing messages is not beneficial. This is because some percentage of messages can converge quickly when updates are

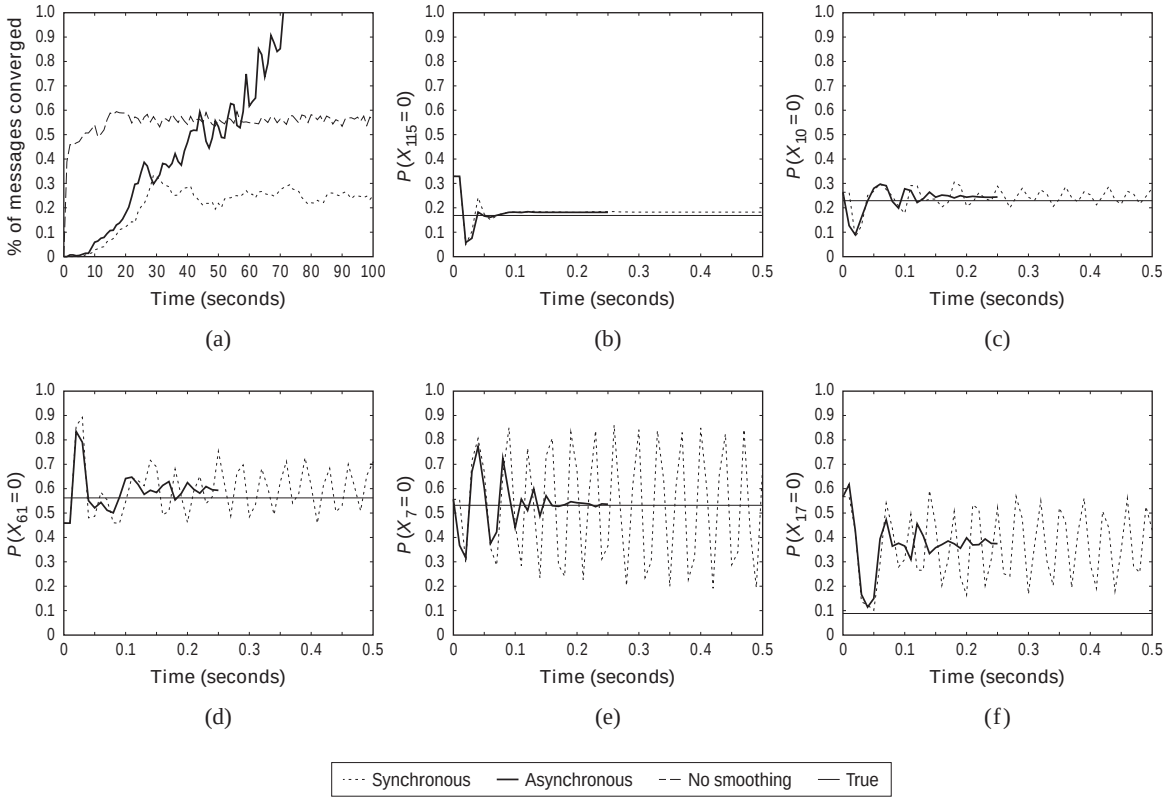


Figure 11.C.1 — Example of behavior of BP in practice on an 11×11 Ising grid. (a) Percentage of messages converged as a function of time for three different BP variants. (b) A marginal where both variants converge rapidly. (c–e) Marginals where the synchronous BP marginals oscillate around the asynchronous BP marginals. (f) A marginal where both variants are inaccurate.

not slowed down by smoothing. However, the overall benefit of damping is evident, and without it the algorithm never converges.

The remaining panels illustrate the progression of the marginal beliefs over the course of the algorithm. (b) shows a marginal where both the synchronous and asynchronous updates converge quite rapidly and are close to the true marginal (thin solid black). Such behavior is atypical, and it comprises only around 10 percent of the marginals in this example. In the vast majority of the cases (almost 80 percent in this example), the synchronous beliefs oscillate around the asynchronous ones ((c)–(e)). In many cases, such as the ones shown in (e), the entropy of the synchronous beliefs is quite significant. For about 10 percent of the marginals (for example (f)), both the asynchronous and synchronous marginals are inaccurate. In these cases, using more informed message schedules can significantly improve the algorithms performance.

These qualitative differences between the BP variants are quite consistent across many random

and real-life models. Typically, the more complex the inference problem, the larger the gaps in performance. For very complex real-life networks involving tens of thousands of variables and multiple cycles, even asynchronous BP is not very useful and more elaborate propagation methods or convergent alternatives must *be adopted.*

11.3.6 Variational Analysis

So far, our discussion of cluster-graph belief propagation has been procedural, motivated purely by similarity to message passing algorithms for cluster trees. Is there any formal justification for this approach? Is there a sense in which we can view this algorithm as providing an approximation to the exact inference task? In this section, we show that cluster-graph belief propagation can be justified using the energy functional formulation of section 11.1. Specifically, the messages passed by cluster-graph belief propagation can be derived from fixed-point equations for the stationary points of an approximate version of the energy functional of equation (11.3). As we will see, this formulation provides significant insight into the generalized belief propagation algorithm. It allows us to understand better the convergence properties of cluster-graph belief propagation and to characterize its convergence points. It also suggests generalizations of the algorithm that have better convergence properties, or that optimize a better approximation to the energy functional.

Our construction will be similar to the one in section 11.2 for exact inference. However, there are important differences that underlie the fact that this algorithm is only an approximate inference algorithm.

factored energy functional

First, the exact energy functional $F[\tilde{P}_\Phi, Q]$ has terms involving the entropy of an entire joint distribution; thus, it cannot be tractably optimized. However, the *factored energy functional* $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ is defined in terms of entropies of clusters and sepsets, each of which can be computed efficiently based purely on local information at the clusters. Importantly, however, unlike for clique trees, $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ is no longer simply a reformulation of the energy functional, but rather an approximation of it.

However, even the factored energy functional cannot be optimized over the space of all marginals $\boldsymbol{Q}$ that correspond to some actual distribution P_Φ. More precisely, consider some cluster graph $\mathcal{U}$; for a distribution P, we define $\boldsymbol{Q}_P = \{P(\boldsymbol{C}_i)\}_{i \in \mathcal{V}_\mathcal{U}} \cup \{P(\boldsymbol{S}_{i,j})\}_{(i-j) \in \mathcal{E}_\mathcal{U}}$. We now define the *marginal polytope* of $\mathcal{U}$ to be

marginal polytope

$$Marg[\mathcal{U}] = \{\boldsymbol{Q}_P : P \text{ is a distribution over } \mathcal{X}\} \tag{11.15}$$

That is, the marginal polytope is the set of all cluster (and sepset) beliefs that can be obtained from marginalizing an actual distribution P. It is called the marginal polytope because it is the set of marginals obtained from the polytope of all probability distributions over $\mathcal{X}$. Unfortunately, not every set of beliefs that correspond to clusters in $\mathcal{U}$ is in the marginal polytope; that is, there are calibrated cluster graph beliefs that do not represent the marginals of any single coherent joint distribution over $\mathcal{X}$ (see exercise 11.2). However, the marginal polytope is a complex object with exponentially many facets. (In fact, the problem of determining whether a set of beliefs is in the marginal polytope can be shown to be $\mathcal{NP}$-hard.) Thus, optimizing a function over the

marginal polytope is a computationally difficult task that is generally as hard as exact inference over the cluster graph. To circumvent these problems, we perform our optimization over the *local consistency polytope*:

local consistency polytope

$$Local[\mathcal{U}] = \left\{ \begin{array}{c} \{\beta_i : i \in \mathcal{V}_\mathcal{U}\} \cup \\ \{\mu_{i,j} : (i\text{–}j) \in \mathcal{E}_\mathcal{U}\} \end{array} \left| \begin{array}{lcll} \mu_{i,j}[\boldsymbol{s}_{i,j}] & = & \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i(\boldsymbol{c}_i) & \forall (i\text{–}j) \in \mathcal{E}_\mathcal{U}, \forall \boldsymbol{s}_{i,j} \in Val(\boldsymbol{S}_{i,j}) \\ 1 & = & \sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) & \forall i \in \mathcal{V}_\mathcal{U} \\ \beta_i(\boldsymbol{c}_i) & \geq & 0 & \forall i \in \mathcal{V}_\mathcal{U}, \boldsymbol{c}_i \in Val(\boldsymbol{C}_i). \end{array} \right. \right\} \tag{11.16}$$

pseudo-marginals

We can think of the local consistency polytope as defining a set of *pseudo-marginal distributions*, each one over the variables in one cluster. The constraints imply that these pseudo-marginals must be calibrated and therefore locally consistent with each other. However, they are not necessarily marginals of a single underlying joint distribution.

Overall, we can write down an optimization problem as follows:

CGraph-Optimize:

Find $\boldsymbol{Q}$
maximizing $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$
subject to

$$\boldsymbol{Q} \in Local[\mathcal{U}] \tag{11.17}$$

☞ **Thus, our optimization problem contains two approximations: We are using an approximation, rather than an exact, energy functional; and we are optimizing it over the space of pseudo-marginals, which is a relaxation (a superspace) of the space of all coherent probability distributions that factorize over the cluster graph.**

In section 11.1, we noted that the energy functional is a lower bound on the log-partition function; thus, by maximizing it, we get better approximations of P_Φ. Unfortunately, the factored energy functional, which is only an approximation to the true energy functional, is not necessarily also a lower bound. Nonetheless, it is still a reasonable strategy to maximize the approximate energy functional, since it may lead to a good approximation of the log-partition function.

This maximization problem directly generalizes CTree-Optimize to the case of cluster graphs. Not surprisingly, we can derive a similar analogue to theorem 11.3, where we characterize the stationary points of this optimization problem as solutions to a set of *fixed-point equations*.

fixed-point equations

Theorem 11.5

A set of beliefs $\boldsymbol{Q}$ is a stationary point of CGraph-Optimize *if and only if for every edge $(i\text{–}j) \in \mathcal{E}_\mathcal{U}$ there are auxiliary factors $\delta_{i \to j}(\boldsymbol{S}_{i,j})$ and $\delta_{j \to i}(\boldsymbol{S}_{j,i})$ so that*

$$\delta_{i \to j} \propto \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in \mathrm{Nb}_i - \{j\}} \delta_{k \to i}. \tag{11.18}$$

and moreover, we have that

$$\begin{array}{rcl} \beta_i & \propto & \psi_i \cdot \prod_{j \in \mathrm{Nb}_i} \delta_{j \to i} \\ \mu_{i,j} & = & \delta_{j \to i} \cdot \delta_{i \to j}. \end{array}$$

The proof is identical to the proof of theorem 11.3.

This theorem shows that we can characterize convergence points of the energy function in terms of the original potentials and messages between clusters. We can, once again, define a procedural variant, in which we initialize $\delta_{i \to j}$, and then iteratively use equation (11.18) to redefine each $\delta_{i \to j}$ in terms of the current values of other $\delta_{k \to i}$. This process is identical (up to a renormalization step) to the update formula we use in CTree-SP-calibrate (algorithm 10.2). Indeed, we defined CGraph-SP-Calibrate, a cluster graph version of CTree-SP-Calibrate, the message passing steps are simply executing this iterative process using the fixed-point equation. Theorem 11.5 shows that convergence points of this procedure are related to stationary points of $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$.

Corollary 11.1

$\boldsymbol{Q}$ is the convergence point of applying CGraph-SP-Calibrate$(\Phi, \mathcal{U})$ *if and only if $\boldsymbol{Q}$ is a stationary point of $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$.*

Due to the equivalence between sum-product and belief update messages, it follows that convergence points of CGraph-BU-Calibrate are also convergence points of CGraph-SP-Calibrate.

Corollary 11.2

At convergence of CGraph-BU-Calibrate*, the set of beliefs is a stationary point of $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$.*

It is tempting to interpret this result as stating that the convergence points of belief propagation are maxima of the factored energy functional. However, there are several gaps between the theorem and this idealized interpretation, which it is important to understand. First, we note that maxima of a function are not necessarily fixed points. In this case, we can verify that $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ is bounded from above, and thus must have a maximum. However, if the maximum is a boundary point (where some of the probabilities in $\boldsymbol{Q}$ are 0), it may not be a fixed point. Fortunately, this situation is rare in practice, and it can be guaranteed not to arise under fairly benign assumptions.

Second, we note that maxima are not the only fixed points of the belief propagation algorithm; minima and saddle points are also fixed points. Intuitively, however, such solutions are not likely to be stable, in the sense that slight perturbations to the messages will drive the process away from them. Indeed, it is possible to show (although this result is outside the scope of this book) that *stable convergence points* of belief propagation are always local maxima of the function.

stable convergence point

The most important limitation of this result, however, is that it does not show that we can reach these maxima by applying belief propagation steps. There is no guarantee that the message passing steps of cluster-graph belief propagation necessarily improve the energy functional: a message passing step may increase or decrease the energy functional. Indeed, as we showed, there are examples where the belief propagation procedure oscillates indefinitely and fails to converge. Even more surprisingly, this problem is not simply a matter of the algorithm being unable to "find" the maximum. One can show examples where the global maximum is not a *stable* convergence point of belief propagation. That is, while it is, in principle, a fixed point of the algorithm, it will never be reached in practice, since even a slight perturbation will give rise to oscillatory behavior.

Nevertheless, this result is of significant importance in several ways. First, it provides us with a declarative semantics for cluster-graph belief propagation in terms of optimization of a target functional. The success of the belief propagation algorithm, when it converges, leads us to hope that the development of new, possibly more convergent, methods to solve the optimization

problem may give rise to good solutions. Second, the declarative view defines the problem in terms of an objective — the factored energy functional — and a set of constraints — the set of locally consistent pseudo-marginals. Both of these are approximations to the ones used in the optimization problem for exact inference. When we view the task from this perspective, some potential directions for improvements become obvious: We can perhaps achieve a better approximation by making our objective a better approximation to the true energy functional, or by tightening our constraints so as to make the constraint space closer to the exact marginal polytope. We will describe some of the extensions based on these ideas; others are mentioned in section 11.7.

11.3.7 Other Entropy Approximations ⋆

The variational analysis of the previous section provides us with a framework for understanding the properties of this type of approximation, and for providing significant generalizations.

11.3.7.1 Motivation

To understand this general framework, consider first the form of the factored energy functional when our cluster graph $\mathcal{U}$ has the form of the Bethe approximation. Recall that in the Bethe approximation graph there are two layers: one consisting of clusters that correspond to factors in Φ, and the other consisting of univariate clusters. When the cluster graph is calibrated, these univariate clusters have the same distribution as the sepsets between them and the factors in the first layer. As such, we can combine together the entropy terms for all the sepsets labeled by X and the associated univariate cluster and rewrite the energy functional, as follows:

Proposition 11.3 *If $\boldsymbol{Q} = \{\beta_\phi : \phi \in \Phi\} \cup \{\beta_i(X_i)\}$ is a calibrated set of beliefs for a Bethe cluster graph $\mathcal{U}$ with clusters $\{\boldsymbol{C}_\phi : \phi \in \Phi\} \cup \{X_i : X_i \in \mathcal{X}\}$, then*

$$\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] = \sum_{\phi \in \Phi} \boldsymbol{E}_{Scope[\phi] \sim \beta_\phi}[\ln \phi] + \sum_{\phi \in \Phi} \boldsymbol{H}_{\beta_\phi}(\boldsymbol{C}_\phi) - \sum_i (d_i - 1)\boldsymbol{H}_{\beta_i}(X_i), \tag{11.19}$$

where $d_i = |\{\phi : X_i \in Scope[\phi]\}|$ is the number of factors that contain X_i.

Note that equation (11.19) is equivalent to the factored energy functional only when $\boldsymbol{Q}$ is calibrated. However, because we are interested only in such cases, we can freely alternate between the two forms for the purpose of finding fixed points of the factored energy functional. Equation (11.19) is the negation of a term known as the *Bethe free energy* in statistical mechanics. The Bethe cluster graph we discussed earlier is a construction that is designed to match the Bethe free energy functional.

Bethe free energy

Why is this reformulation useful? Recall that, in our discussion of generalized cluster graphs, we required the running intersection property. This property has two important implications. First is that the set of clusters that contain some variable X are connected; hence, the marginal over X will be the same in all of these clusters at the calibration point. Second is that there is no cycle of clusters and sepsets all of which contain X. We motivated this assumption by noting that it prevents us from allowing information about X to cycle endlessly through a loop. This new formulation provides a more formal justification. As we can see, if the variable X_i

appears in d_i clusters in the cluster graph, then it appears in an entropy term with a positive sign exactly d_i times. Owing to the running intersection property, the number of sepsets that contain X_i is $d_i - 1$ (the number of edges in a tree with k vertices is $k-1$), so that X_i appears in an entropy term with a negative sign exactly $d_i - 1$ times. In this case, the entropy of X_i appears with positive sign d_i times, and with negative sign $d_i - 1$ times, so overall it is counted exactly once.

This reformulation suggests a much more general class of entropy approximations. We can construct define a set of *regions* $\mathbf{R}$, each with its own scope $\boldsymbol{C}_r$ and its own *counting number* κ_r. We can now define the following *weighted approximate entropy*:

counting number

weighted approximate entropy

$$\tilde{\boldsymbol{H}}^{\boldsymbol{\kappa}}_{\boldsymbol{Q}}(\mathcal{X}) = \sum_r \kappa_r \boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r). \tag{11.20}$$

Example 11.2

Bethe cluster graph

The simple Bethe cluster graph *of section 11.3.5.2 fits easily into this new framework. The construction has two levels of regions: a set of "large" $\mathbf{R}^+$, where each $r \in \mathbf{R}^+$ contains multiple variables, and singleton regions containing the individual variables $X_i \in \mathcal{X}$. Both types of regions have counting numbers: κ_r for $r \in \mathbf{R}^+$ and κ_i for $X_i \in \mathcal{X}$. All factors in Φ are assigned only to large regions, so that $\psi_i = 1$ for all i. We use Nb_r to denote the set $\{X_i \in \boldsymbol{C}_r\}$, and Nb_i to denote the set $\{r : X_i \in \boldsymbol{C}_r\}$.*

To capture exactly the Bethe free energy, we set each large region to have a counting number of 1, and each singleton region corresponding to X_i to have a counting number of $1 - d_i$ where d_i is the number of large regions that contain X_i in their scope. We see that in this construction the region graph energy functional is identical to the Bethe free energy of equation (11.19). ■

However, this framework also allows us to capture much richer constructions.

Example 11.3

Consider again the example of figure 11.8a. As we discussed in section 11.3.5.3, this cluster graph has the benefit of maintaining the pairwise correlations between all pairs of variables when passing messages between clusters. Unfortunately, it is not a legal cluster graph, since it does not satisfy the running intersection property. We can obtain another perspective on the problem with this cluster graph by examining the energy functional associated with it:

$$\begin{aligned}\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] \quad = \quad & \boldsymbol{E}_{\beta_1}[\ln \phi_1(A,B,C)] + \boldsymbol{E}_{\beta_2}[\ln \phi_2(B,C,D)] + \boldsymbol{E}_{\beta_3}[\ln \phi_3(A,C,D)] \\ & + \boldsymbol{H}_{\beta_1}(A,B,C) + \boldsymbol{H}_{\beta_2}(B,C,D) + \boldsymbol{H}_{\beta_3}(A,C,D) \\ & - \boldsymbol{H}_{\beta_4}(B,C) - \boldsymbol{H}_{\beta_5}(A,C) - \boldsymbol{H}_{\beta_6}(C,D).\end{aligned}$$

As we can see, the variable C appears in three clusters and three sepsets. As a consequence, the counting number of C in the energy functional is 0. This means that we are undercounting the entropy of C in the approximation. Indeed, as we discussed, this cluster graph does not satisfy the running intersection property. Thus, we considered modifying the graph by removing C from one of the sepsets. However, if we consider this problem from the perspective of the energy functional, we can deal with the problem by adding another factor β_7 that has C as its scope. If we add $\boldsymbol{H}_{\beta_7}(C)$ to the energy functional we solve the undercounting problem. This results in a modified energy functional

$$\begin{aligned}\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] \quad = \quad & \boldsymbol{E}_{\beta_1}[\ln \phi_1(A,B,C)] + \boldsymbol{E}_{\beta_2}[\ln \phi_2(B,C,D)] + \boldsymbol{E}_{\beta_3}[\ln \phi_3(A,C,D)] \\ & + \boldsymbol{H}_{\beta_1}(A,B,C) + \boldsymbol{H}_{\beta_2}(B,C,D) + \boldsymbol{H}_{\beta_3}(A,C,D) + \boldsymbol{H}_{\beta_7}(C) \\ & - \boldsymbol{H}_{\beta_4}(B,C) - \boldsymbol{H}_{\beta_5}(A,C) - \boldsymbol{H}_{\beta_6}(C,D).\end{aligned}$$

This is simply an instance of our weighted entropy approximation, with seven regions: the three triplets, the three pairs, and the singleton C. ■

This perspective provides a clean and simple framework for proposing generalizations to the class of approximations defined by the cluster graph framework. Of course, to formulate our optimization problem fully, we need to define the constraints and construct algorithms that solve the resulting optimization problems. We now address these issues in the context of two different classes of weighted entropy approximations.

11.3.7.2 Convex Approximations

One of the biggest problems with the objective used in standard loopy BP is that it gives rise to a nonconvex optimization problem. In fact, the objective often has multiple local optima. These properties make the optimization hard and the answers nonrobust. However, a different choice of counting numbers can lead to a concave optimization objective, and hence to a convex optimization problem. Such problems are much easier to solve using a range of algorithms, and the solutions offer a satisfying guarantee of optimality. We first define the class of convex BP objectives and then describe one solution algorithm.

We focus our discussion on energy functionals whose structure uses the two-layer Bethe cluster graph structure of example 11.2, but where the counting numbers are different. To preserve the desired semantics of the counting numbers, we require:

$$\kappa_i = 1 - \sum_{r \in \mathrm{Nb}_i} \kappa_r, \tag{11.21}$$

ensuring that the total counting number of terms involving the entropy of X_i is precisely 1. When we define $\kappa_r = 1$ for all $r \in \mathbf{R}$, this constraint implies the counting numbers in the Bethe free energy.

We now introduce the following condition on the counting numbers:

Definition 11.4 convex counting numbers

We say that a vector of counting numbers κ_r is convex *if there exist* nonnegative *numbers ν_r, ν_i, and $\nu_{r,i}$ such that:*

$$\begin{array}{rcll} \kappa_r & = & \nu_r + \sum_{i \,:\, X_i \in \boldsymbol{C}_r} \nu_{r,i} & \textit{for all } r \\ \kappa_i & = & \nu_i - \sum_{r \,:\, X_i \in \boldsymbol{C}_r} \nu_{r,i} & \textit{for all } i \end{array} \tag{11.22}$$

■

Assuming that we have a set of convex counting numbers, we can rewrite the weighted approximate entropy of equation (11.20) as:

$$\begin{aligned} &\sum_r \kappa_r \boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r) + \sum_i \kappa_i \boldsymbol{H}_{\beta_i}(X_i) = \\ &\qquad \sum_r \nu_r \boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r) + \sum_{r, X_i \in \boldsymbol{C}_r} \nu_{r,i} (\boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r) - \boldsymbol{H}_{\beta_i}(X_i)) + \sum_i \nu_i \boldsymbol{H}_{\beta_i}(X_i). \end{aligned} \tag{11.23}$$

Importantly, when the beliefs satisfy the marginal-consistency constraints, the terms in the second summation can be rewritten as conditional entropies:

$$\boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r) - \boldsymbol{H}_{\beta_i}(X_i) = \boldsymbol{H}_{\beta_r}(\boldsymbol{C}_r \mid X_i).$$

Plugging this result back into equation (11.23), we obtain an objective that is a summation of terms each of which is either an entropy or a conditional entropy, all with positive coefficients. Because both entropies and conditional entropies are convex, we obtain the following result:

Proposition 11.4 *The function in equation (11.23) is a concave function for any set of beliefs $\boldsymbol{Q}$ that satisfies the marginal consistency constraints.*

concave over constraints

convex entropy

This type of objective function is called *concave over the constraints*, since it is not generally concave, but it is concave over the subspace that satisfies the constraints of our optimization problem. An entropy as in equation (11.23) that uses convex counting numbers is called a *convex entropy*.

Assuming that the potentials are all strictly positive, we can now conclude that the optimization problem CGraph-Optimize with convex counting numbers is a convex optimization problem that has a unique global optimum.

Convex optimization problems can, in principle, be solved by a range of different algorithms, all of which guarantee convergence to the unique global optimum. However, the basic optimization problem can easily get intractably large. Recall that to formulate our optimization space, we need to introduce an optimization variable for each assignment of values to each cluster in our cluster graph, and a constraint for each assignment of values to each sepset in the graph.

Example 11.4 *Consider a grid-structured network corresponding to a modestly sized 500×500 image, where each pixel can take 100 values. The structure of the graph is a pairwise network, with approximately $2 \times 250,000$ clusters (pairwise edges), each of which can take $100 \times 100 = 10,000$ values. The total number of variables is therefore $500,000 \times 10,000 = 5 \times 10^9$, an unmanageable number for most optimizers.* ■

Fortunately, due to the convexity of this problem, we have that strong duality holds (see appendix A.5.4), and therefore we can find a solution to this problem by solving its dual. The message passing algorithms that we derive from the Lagrange multipliers are one method that we can use for solving the dual. (For example, exercise 11.17 provides one message passing algorithm for a Bethe cluster graph with general counting numbers.) However, the message passing algorithms are not directly optimizing the objective. Rather, they characterize the optimum using a set of fixed-point equations, and attempt to converge to the optimum by iterating through these equations. This process is generally not guaranteed to achieve the optimum, even when the problem is convex. Again, we can consider using other optimization algorithms over the dual problem. However, a message passing approach has some important advantages, such as modularity and efficiency.

Fortunately, a careful reformulation of the message passing scheme can be shown to guarantee convergence to the global optimum. This reformulation is different for synchronous and asynchronous message passing. We present the asynchronous version, which is simpler and also likely to be more efficient in practice.

The algorithm, shown in algorithm 11.2, uses the following quantities in its computations:

$$\hat{\nu}_i = \nu_i + \sum_{r \in \mathrm{Nb}_i} \nu_r; \qquad \hat{\nu}_{i,r} = \nu_r + \nu_{i,r}. \tag{11.24}$$

In each message passing iteration, it traverses the variables X_i in a round-robin fashion; for each X_i, it computes two sets of messages: incoming messages $\delta_{r \to i}(X_i)$ from regions to variables,

Algorithm 11.2 Convergent message passing for Bethe cluster graph with convex counting numbers

Procedure Convex-BP-Msg (
  $\psi_r(\boldsymbol{C}_r)$ // set of initial potentials
  $\sigma_{i \to r}(\boldsymbol{C}_r)$ // Current node-to-region messages
)

for $i = 1, \ldots, n$
  // Compute incoming messages from neighboring regions to X_i
  for $r \in \mathrm{Nb}_i$
    $\delta_{r \to i}(X_i) \leftarrow \sum_{\boldsymbol{C}_r - X_i} \left(\psi_r(\boldsymbol{C}_r) \prod_{j \in \mathrm{Nb}_r - \{i\}} \sigma_{j \to r}(\boldsymbol{C}_r) \right)^{\frac{1}{\hat{\nu}_{i,r}}}$
    // Compute beliefs for X_i, renormalizing to avoid numerical underflows
  $\beta_i(X_i) \leftarrow \frac{1}{Z_{X_i}} \prod_{r \in \mathrm{Nb}_i} (\delta_{r \to i}(X_i))^{\hat{\nu}_{i,r}/\hat{\nu}_i}$
    // Compute outgoing messages from X_i to neighboring regions
  for $r \in \mathrm{Nb}_i$
    $\sigma_{i \to r}(\boldsymbol{C}_r) \leftarrow \left(\psi_r(\boldsymbol{C}_r) \prod_{j \in \mathrm{Nb}_r - \{i\}} \sigma_{j \to r}(\boldsymbol{C}_r) \right)^{-\frac{\nu_{i,r}}{\hat{\nu}_{i,r}}} \left(\frac{\beta_i(X_i)}{\delta_{r \to i}(X_i)} \right)^{\nu_r}$
return $\{\sigma_{i \to r}(\boldsymbol{C}_r)\}_{i, r \in \mathrm{Nb}_i}$

factor graph

and outgoing messages $\sigma_{i \to r}(\boldsymbol{C}_r)$ from variables to regions (essentially passing messages over the *factor graph*). The overall process is initialized (in the first message passing iteration) by setting $\sigma_{i \to r} = 1$. This algorithm is guaranteed to converge to the global maximum of our convex energy functional.

This derivation applies to any set of convex counting numbers, leaving open the question of which counting numbers are likely to be give the best approximation. Although there is currently no theoretical analysis answering this question, intuitively, we might argue that we want the counting numbers for different regions to be as close as possible to uniform. This intuition is also supported by the fact that the Bethe approximation, which sets all $\kappa_r = 1$, obtains very high-quality approximations when it converges. Thus, we can try to select nonnegative coefficients ν_i, ν_r, and $\nu_{i,r}$ for which κ_r and κ_i, defined via equation (11.22), satisfy equation (11.21) and minimize

$$\sum_{r \in \mathbf{R}^+} (\kappa_r - 1)^2. \tag{11.25}$$

TRW

Other choices are also possible. For example, the *tree-reweighted belief propagation (TRW)* algorithm computes convex counting numbers for a pairwise Markov network using the following process: We first define a probability distribution ρ over trees $\mathcal{T}$ in the network, such that each edge in the pairwise network is present in at least one tree. This distribution defines a set of weights:

$$\begin{aligned} \kappa_i &= -\textstyle\sum_{\mathcal{T} \ni X_i} \rho(\mathcal{T}) \\ \kappa_{i,j} &= \textstyle\sum_{\mathcal{T} \ni (X_i, X_j)} \rho(\mathcal{T}) \end{aligned} \tag{11.26}$$

convex counting numbers

This computation results in a set of *convex counting numbers* (see exercise 11.18). Preliminary results suggest that the TRW counting numbers and the ones derived from optimizing equation (11.25) appear to achieve similar performance in practice.

However, the comparison to standard (Bethe-approximation) BP is less clear. **When standard BP converges, it generally tends to produce better results than the convex counterparts, and almost universally it converges much faster. Conversely, when standard BP does not converge, the convex algorithms have an advantage;** but, as we discuss in box 11.B, there are many tricks we can use to improve the convergence of BP, so it is not clear how often nonconvergence is a problem. One setting where a convergent algorithm can have important benefits is in those settings (chapter 19 and chapter 20) where we generally learn the model using iterative, hill-climbing methods that use inference in the inner loop for tasks such as gradient computations. There, the use of a nonconvergent algorithm for computing the gradient can severely destabilize the learning algorithm. In other settings, however, the decision of whether to use standard or convex BP is one of approximate optimization of a pretty good (although still approximate) objective, versus exact optimization of an objective that is generally not as good. The right decision in this trade-off is not clear, and needs to be made specifically for the target application.

11.3.7.3 Region Graph Approximations

As illustrated in example 11.3, a very different motivation for using an objective based on different counting numbers is to improve the quality of the approximation by better capturing interactions between variables. As we showed in this example, we can use the notion of a weighted entropy approximation to define a (hopefully) better approximation to the entropy. Of course, to specify the optimization problem fully, we also need to specify the constraints. In this example, it is fairly straightforward to do so: we want $\beta_7(C)$ to be consistent with the marginal probability of C in one of the other beliefs that mention C. Now, we have an optimization problem that seems to solve the problem we set out to solve: It can compute beliefs on each of the original factors while maintaining consistency at the level of each pairwise marginal shared among these factors.

However, the new optimization problem we defined is not one that corresponds to a cluster graph. To see this, notice that β_7 appears in the role of a cluster. But, if it is a cluster, it would have to be connected to one of the other factors by a sepset with scope C, which would require an additional term in the energy functional associated with this cluster graph. Thus, it is not immediately clear how we would go about optimizing the new modified functional.

We now discuss a general framework that defines the form of the optimization objective and the constraints for constructions that capture higher-level interactions between the variables. We also describe a message passing algorithm that can be used to find fixed points of this optimization problem.

Region Graphs The basic structure we consider is similar to a cluster graph, but unlike cluster graphs we no longer distinguish two types of vertices (clusters and sepsets). Rather, we can have a more deeply nested hierarchy of regions, related by containment.

Definition 11.5

region graph

A region graph $\mathcal{R}$ *is a directed graph over a set of vertices* $\mathbf{R}$*. Each vertex* r *is called a* region *and*

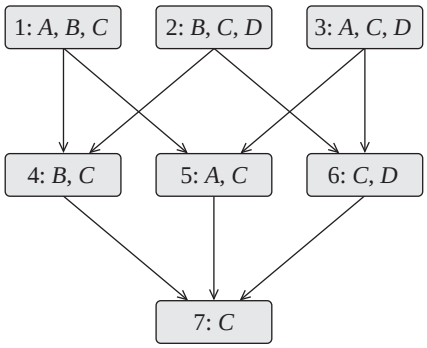


Figure 11.9 An example of simple region graph

is associated with a distinct set of random variables $\boldsymbol{C}_r$. *Whenever there is an arc from a region* r_i *to a region* r_j *we require that* $\mathit{Scope}[r_i] \supset \mathit{Scope}[r_j]$. *Regions that have no incoming edges are called* top regions.

counting numbers

Each region is associated with a counting number $\kappa_r \in \mathbb{R}$. *Note that the counting number may be negative, positive, or zero.* ■

Because containment is transitive, we have that if there is a directed path from r to r', then $\mathit{Scope}[r'] \subset \mathit{Scope}[r]$. Thus, a region graph is acyclic.

To define the energy term in the free energy, we must assign our original factors in Φ to regions in the region graph. Here, because different regions are counted to different extents, it is useful to allow a factor to be assigned to more than one region. Thus, for each $\phi \in \Phi$ we have a set of regions $\alpha(\phi) \subset \mathbf{R}$ such that $\mathit{Scope}[\phi] \subseteq \boldsymbol{C}_r$. This property is analogous to the *family-preservation property* for cluster graphs. Throughout this book, we assume without loss of generality that any $r \in \alpha(\phi)$ is a top region.

family preservation

We are now ready to define the energy functional associated with a region graph:

$$\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] = \sum_r \kappa_r \boldsymbol{E}_{\boldsymbol{C}_r \sim \beta_r}[\ln \psi_r] + \tilde{\boldsymbol{H}}^{\boldsymbol{\kappa}}_{\boldsymbol{Q}}(\mathcal{X}), \tag{11.27}$$

where ψ_r is defined as:

$$\psi_r = \prod_{\phi \in \Phi \,:\, r \in \alpha(\phi)} \phi.$$

As with cluster graphs, a region graph defines a set of beliefs, one per region. We use the notation $\beta_r(\boldsymbol{C}_r)$ to denote the belief associated with the region r over the set of variables $\boldsymbol{C}_r = \mathit{Scope}[r]$.

Figure 11.9 demonstrates the region graph construction for the approximation of example 11.3. This example contains three regions that correspond to the initial factors in the distribution we want to approximate. The lower set of regions are the pairwise intersections between the three factors. The lowest region is associated with the variable C.

Whereas the counting numbers specify the energy functional, the graph structure specifies the constraints over the beliefs that we wish to associate with the regions. In particular, we

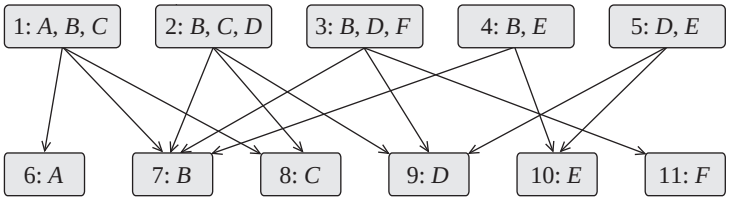


Figure 11.10 The region graph corresponding to the Bethe cluster graph of figure 11.7a

want the beliefs to satisfy the calibration constraints that are implied by the edges in the region graph structure.

Definition 11.6

region graph calibration

Given a region graph, a set of region beliefs is calibrated *if whenever $r \to r'$ appears in the region graph then*

$$\sum_{\boldsymbol{C}_r - \boldsymbol{C}_{r'}} \beta_r(\boldsymbol{C}_r) = \beta_{r'}(\boldsymbol{C}'_r) \tag{11.28}$$

■

The region graph structure provides a very general framework for specifying energy-functional optimization problems. The set of regions and their counting numbers tell us which components appear in the energy functional, and with what weight. The arcs in the region graph tell us which consistency constraints we wish to enforce. We can choose which consistency constraints we want to enforce by adding or removing regions or edges between them; we note that this can be done without affecting the energy functional by simply giving the regions introduced a counting number of 0. The more edges we have, the more constraints we require regarding the calibration of different beliefs.

The region graph framework is very general, and it can encode a broad spectrum of optimization objectives and constraints. However, not all such formulations are equally reasonable as approximations to our true objective, which is the exact energy functional of equation (11.3). The following requirement captures some of the essential properties that make a region graph construction suitable for this purpose.

Definition 11.7

For each variable X_i, let $\mathbf{R}_i = \{r : X_i \in Scope[r]\}$; for each factor $\phi \in \Phi$, let $\mathbf{R}_\phi = \{r : Scope[\phi] \subseteq \boldsymbol{C}_r\}$. We now define the following conditions for a region graph:

- variable connectedness*: for every variable X_i, the set $\mathbf{R}_i$ forms a single connected component;*
- factor connectedness*: for every factor ϕ, the set $\mathbf{R}_\phi$ forms a single connected component;*
- factor preservation*:*

$$\sum_{r \in \alpha(\phi)} \kappa_r = 1.$$

- running intersection*: for every variable X_i,*

$$\sum_{r \in \mathbf{R}_i} \kappa_r = 1.$$

■

The connectedness requirement for variables ensures that the beliefs about any individual variable X_i in any calibrated region graph will be consistent for all beliefs that contain X_i. The counting number condition for variables ensures that we are not not overcounting or undercounting the entropy of X_i. Together, these conditions extend the running intersection property, ensuring that the subgraph containing X_i is connected and that X_i is "counted" once in total. We note that, like the running intersection property, this requirement does not address the extent to which we count the contribution associated with the interactions between pairs or larger subsets of variables.

The factor-preservation condition for factors ensures that, when we sum up the energy terms of the different regions, each factor is counted exactly once in total. As we will see, this ensures that a calibrated region graph will still encode our original distribution P_Φ. Finally, the connectedness condition for factors ensures that we cannot double-count contributions of our initial factors: that is, a factor cannot "flow" around a loop.

An examination confirms that all parts of the region graph condition hold both for the Bethe region graph and for the region graph of figure 11.9.

How do we construct a valid region graph? One approach is based on the following simple recursive construction for the counting numbers. We first define, for a region r,

$$\mathbf{Up}(r) = \{r' \ : \ (r' \to r) \in \mathcal{E}_\mathcal{R}\}$$

to be the set of regions that are directly upwards of r; similarly, we define

$$\mathbf{Down}(r) = \{r' \ : \ (r \to r') \in \mathcal{E}_\mathcal{R}\}.$$

We also define the *upward closure* of r to be the set $\mathbf{Up}^*(r)$ of all the regions from which there is a directed path to r, and the *downward closure* $\mathbf{Down}^*(r)$ to be all the regions that can be reached by a directed path from r; finally, we define $\mathbf{Down}^+(r) = \{r\} \cup \mathbf{Down}^*(r)$.[2]

We can now define the counting numbers recursively, using the following formula:

$$\kappa_r = 1 - \sum_{r' \in \mathbf{Up}^*(r)} \kappa_{r'}. \tag{11.29}$$

This condition ensures that the sum of the counting numbers of r and all of the regions above it will be 1. Intuitively, we can think of the counting number of the region r as correcting for overcounting or undercounting of the weight of the scope of r by regions above it. Now, assume that our region graph is structured so that, for each variable X_i, there is a unique region r_i such that every other region whose scope contains X_i is an ancestor of r_i. Then, we are guaranteed that both the connectedness and the counting number condition for X_i hold. We can similarly require that for any factor ϕ, there is a unique region r_ϕ such that any other region whose scope contains *Scope*$[\phi]$ is an ancestor of r_ϕ. This construction guarantees the requirements for factor connectedness and counting numbers.

It is easy to see that the Bethe region graph of example 11.2 satisfies both of these conditions. Moreover, this process of guaranteeing that a unique minimal region exists for each X_i is essentially what we did in example 11.3 to produce a valid region graph.

saturated region graph

These conditions provide us with a simple strategy for constructing a *saturated region graph*.

2. Some of the literature on region graphs use the terminology of parents and children of regions. To avoid the confusion with similar terminology in Bayesian networks, we use the terms *up* and *down* here.

Algorithm 11.3 Algorithm to construct a saturated region graph

```
    Procedure Build-Saturated-Region-Graph (
      R    // a set of initial regions
    )
1     repeat
2       For any r_1, r_2 ∈ R
3         Z ← Scope[r_1] ∩ Scope[r_2]
4         if Z ≠ ∅
5           and R does not contain a region with scope Z then
6             create region r with Scope[r] = Z
7             R ← R ∪ {r}
8       Until convergence
9       Initialize R as an empty graph with R as vertices
10      for each r_1 ≠ r_2 ∈ R
11        if Scope[r_2] ⊂ Scope[r_1] and
12        not exist r_3 ∈ R such that Scope[r_2] ⊂ Scope[r_3] ⊂ Scope[r_1] then
13            add an arc r_1 → r_2 to the region graph R
14      return R
```

We start with initial set of regions. Often, these regions will be the initial factors in P_Φ, although we can decide to work with bigger regions that capture some more global interactions. We then extend this set of regions into a valid region graph, where our goal is to represent appropriately any subset of variables that is shared by some of the regions. We therefore expand the set of regions to be closed under intersections. We connect these regions so that the upward closure of each region contains all of its supersets. The full procedure is shown in algorithm 11.3. Unlike the Bethe approximation, this region graph maintains the consistency of higher-order marginals. The example of figure 11.9 is an example of running this procedure on the original set of regions $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, C, D\}$. As our previous discussion suggests, this procedure guarantees a region graph that satisfies the region graph condition.

Belief Propagation in Region Graphs Given a region graph, we are faced with the task of optimizing the free energy associated with its structure:

RegionGraph-Optimize:

Find $\boldsymbol{Q} = \{\beta_r : i \in \mathcal{V}_{\mathcal{R}}\}$
maximizing $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$
subject to

$$\sum_{\boldsymbol{C}_{r'} - \boldsymbol{C}_r} \beta_{r'}(\boldsymbol{c}_{r'}) = \beta_r(\boldsymbol{c}_r) \tag{11.30}$$

$$\forall r \in \mathcal{V}_{\mathcal{R}}, \forall r' \in \mathbf{Up}(r), \forall \boldsymbol{c}_r \in Val(\boldsymbol{C}_r)$$

$$\sum_{\boldsymbol{C}_r} \beta_r(\boldsymbol{c}_r) = 1 \qquad \forall r \in \mathcal{V}_{\mathcal{R}} \tag{11.31}$$

$$\beta_r(\boldsymbol{c}_r) \geq 0 \qquad \forall r \in \mathcal{V}_{\mathcal{R}}, \boldsymbol{c}_r \in Val(\boldsymbol{C}_r). \tag{11.32}$$

Our strategy for devising algorithms for solving this optimization problem is similar to the approach we took in section 11.3.6. Using the method of Lagrange multipliers, we characterize the stationary points of the target function (given the constraints) as a set of fixed-point equations. We then find an iterative algorithm that attempts to reach such a stationary point.

We first characterize the fixed point via the Lagrange multipliers. As before, we form a Lagrangian by introducing terms for each of the constraints: from equation (11.30), we obtain a Lagrange multiplier $\lambda_{r,r'}(\boldsymbol{c}_r)$ for every pair $r' \in \mathbf{Up}(r)$ and every $\boldsymbol{c}_r \in Val(\boldsymbol{C}_r)$; from equation (11.31), we obtain a Lagrange multiplier λ_r for every r and every $\boldsymbol{c}_r \in Val(\boldsymbol{C}_r)$; as before, we assume that we are dealing with interior fixed points only, and so do not have to worry about the inequality constraint. We differentiate the Lagrangian relative to each of the region beliefs $\beta_r(\boldsymbol{c}_r)$, and obtain the following set of *fixed-point equations*:

fixed-point equations

$$\kappa_r \ln \beta_r(\boldsymbol{c}_r) = \lambda_r + \kappa_r \ln \psi_r(\boldsymbol{c}_r) - \sum_{r' \in \mathbf{Up}(r)} \lambda_{r,r'}(\boldsymbol{c}_r) + \sum_{r' \in \mathbf{Down}(r)} \lambda_{r',r}(\boldsymbol{c}_{r'}) - \kappa_r. \tag{11.33}$$

For regions for which $\kappa_r \neq 0$, we can rewrite this equation to conclude that:

$$\beta_r(\boldsymbol{c}_r) = \frac{1}{Z_r} \psi_r(\boldsymbol{c}_r) \left(\frac{\prod_{r' \in \mathbf{Down}(r)} \delta_{r \to r'}(\boldsymbol{c}_{r'})}{\prod_{r' \in \mathbf{Up}(r)} \delta_{r' \to r}(\boldsymbol{c}_r)} \right)^{1/\kappa_r}. \tag{11.34}$$

From this equality, one can conclude the following result:

Theorem 11.6 *Assume that our region graph satisfies the family preservation property. Then, at fixed points of the* RegionGraph-Optimize *optimization problem, we have that:*

$$P_\Phi(\mathcal{X}) \propto \prod_r (\beta_r)^{\kappa_r}. \tag{11.35}$$

The proof is derived from a simple cancellation of messages in the different terms (see exercise 11.16).

This result tells us that we can reparameterize the initial distribution P_Φ in terms of the final beliefs obtained as fixed points of the region graph optimization problem. It tells us that we can represent the distribution in terms of a calibrated set of beliefs for the individual regions. This

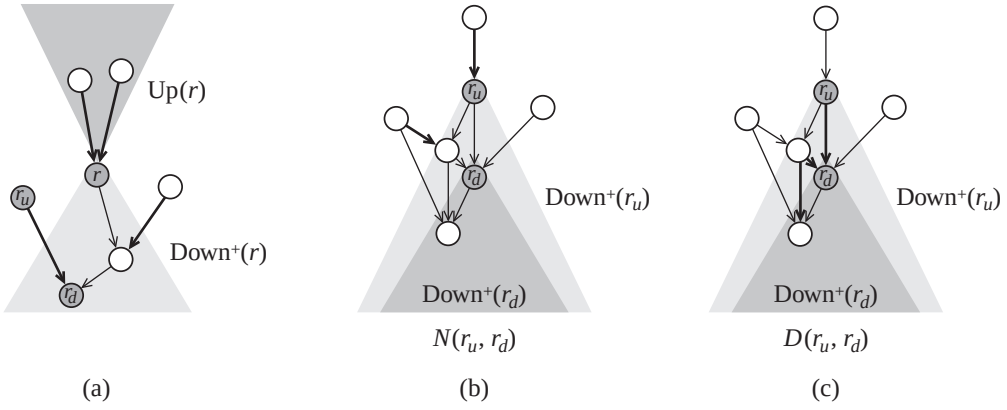


Figure 11.11 The messages participating in different region graph computations. Participating messages are marked with thicker arrows. (a) The computation of the beliefs $\beta_r(\boldsymbol{C}_r)$; (b) The set $N(r_u, r_d)$ participating in the computation of the message $\delta_{r_u \to r_d}$; (c) The set $D(r_u, r_d)$ participating in the computation of the message $\delta_{r_u \to r_d}$.

result is a very powerful one, because it holds for any set of counting numbers that satisfies the family preservation property — a very large class. Of course, this result only shows that any fixed point is a reparameterization of the distribution, but not that such a reparameterization exists. However, under our assumption that all of the initial factors in Φ are strictly positive, one can show that such a fixed point, and hence a corresponding reparameterization, necessarily exists.

message passing algorithm

As we can see, unlike the case of cluster graphs, the fixed-point equations for region graphs are more involved, and do not lead directly to an elegant *message passing algorithm*. Indeed, the derivation of the update rules from the Lagrange multipliers often involves multiple steps of algebraic manipulation. (Although such derivations are possible in restricted cases; see exercise 11.17 and exercise 11.19.) In the remainder of this section, we present, without derivation, one set of update rules that can be derived from equation (11.34), in the specific case where the counting numbers are as presented in equation (11.29).

The basic idea is similar to the message update used in cluster graphs. There, each sepset carried messages sent by one of the clusters through the sepset to the neighboring cluster. We can think of that as a message from the cluster to the sepset. The analog of this in region graph is a message from a region to a region below it. Thus, for each pair $r_1 \to r_2$ in the region graph we will have a message $\delta_{r_1 \to r_2}$ whose scope is $Scope[r_2]$. All messages are associated with "downward" edges of the form $r_1 \to r_2$, but they are used to define the beliefs and messages of regions that contain them.

The definitions of the messages and the beliefs are somewhat involved. We begin by defining

the beliefs of a region as a function of these messages, which is somewhat simpler:

$$\beta_r(\boldsymbol{C}_r) = \psi_r(\boldsymbol{C}_r) \left(\prod_{r_u \in \mathbf{Up}(r)} \delta_{r_u \to r}(\boldsymbol{C}_r) \right) \left(\prod_{r_d \in \mathbf{Down}^*(r)} \prod_{r_u \in \mathbf{Up}(r_d) - \mathbf{Down}^+(r)} \delta_{r_u \to r_d}(\boldsymbol{C}_{r_d}) \right). \tag{11.36}$$

In other words, the belief of a region is the product of three groups of terms. The first two are very natural: the initial beliefs ψ_r (1 for all regions except the top ones) and the messages from its upward regions. The last group contain all messages sent to regions below the region r from regions other than the region r itself and regions below it. In other words, these are messages from "external" sources to regions below the region r; see figure 11.11a. Thus, the beliefs of the region are not influenced by messages it sends down, but only by messages sent to it or its subsets from other regions.

Again, it is instructive to compare this definition to our definition of beliefs in cluster graphs. In cluster graphs, the belief over a sepset is the product of messages from the neighboring clusters. These messages correspond in our case to messages from upward regions. The belief over a cluster $\boldsymbol{C}$ is the product of its initial potential and messages sent from neighboring clusters to the sepsets adjacent to $\boldsymbol{C}$. The sepsets correspond to the regions in $\mathbf{Down}^+(\boldsymbol{C})$; the message sent by another clique $\boldsymbol{C}'$ to this sepset corresponds to messages sent by an "external" source to a region in $\mathbf{Down}^+(\boldsymbol{C})$.

We now move to defining the computation of a message from r_u to r_d, also illustrated in figure 11.11b,c:

$$\delta_{r_u \to r_d}(\boldsymbol{C}_{r_d}) = \frac{\sum_{\boldsymbol{C}_{r_u} - \boldsymbol{C}_{r_d}} \psi_{r_u}(\boldsymbol{C}_{r_u}) \prod_{r_1 \to r_2 \in N(r_u, r_d)} \delta_{r_1 \to r_2}(\boldsymbol{C}_{r_2})}{\prod_{r_1 \to r_2 \in D(r_u, r_d)} \delta_{r_1 \to r_2}(\boldsymbol{C}_{r_2})}. \tag{11.37}$$

The numerator involves the initial factor assigned to the region, and a product of messages associated with the set of edges

$$N(r_u, r_d) = \{(r_1 \to r_2) \in \mathcal{E}_{\mathcal{R}} : r_1 \notin \mathbf{Down}^+(r_u), r_2 \in \mathbf{Down}^+(r_u) - \mathbf{Down}^+(r_d)\}.$$

This set contains edges from sources "external" to r_u that are outside the scope of influence of r_d; that is, they either enter r_u directly, or enter regions below r_u that are not below r_d. The denominator involves a product of the messages in the set:

$$D(r_u, r_d) = \{(r_1 \to r_2) \in \mathcal{E}_{\mathcal{R}} : r_1 \in \mathbf{Down}^+(r_u) - \mathbf{Down}^+(r_d), r_2 \in \mathbf{Down}^+(r_d)\}.$$

This set counts information that would be passed from r_u to regions below r_d *indirectly* — not through r_d. We want to divide by these messages, since otherwise the same information would be incorporated multiple times into the beliefs and the messages.

Example 11.5 *We now return to the region graph of figure 11.9 that corresponds to the potential function we discussed in example 11.3.*

Applying equation (11.36) to this example, we get a set of equation representing the potentials as function of the initial factors and the messages:

$$\begin{aligned}
\beta_1 &= \psi_1 \delta_{2\to4}\delta_{3\to5}\delta_{6\to7} \\
\beta_2 &= \psi_2 \delta_{1\to4}\delta_{3\to6}\delta_{5\to7} \\
\beta_3 &= \psi_3 \delta_{1\to5}\delta_{2\to6}\delta_{4\to7} \\
\beta_4 &= \delta_{1\to4}\delta_{2\to4}\delta_{5\to7}\delta_{6\to7} \\
\beta_5 &= \delta_{1\to5}\delta_{3\to5}\delta_{4\to7}\delta_{6\to7} \\
\beta_6 &= \delta_{2\to6}\delta_{3\to6}\delta_{4\to7}\delta_{5\to7} \\
\beta_7 &= \delta_{4\to7}\delta_{5\to7}\delta_{6\to7}.
\end{aligned}$$

Applying equation (11.37), we can construct the messages. For example,

$$\delta_{4\to7} = \sum_B \delta_{1\to4}\delta_{2\to4}.$$

One easy way to derive this message directly is to use the marginal consistency constraint:

$$\beta_7 = \sum_B \beta_4.$$

Plugging in the expanded form of the two beliefs, we get

$$\delta_{4\to7}\delta_{5\to7}\delta_{6\to7} = \sum_B \delta_{1\to4}\delta_{2\to4}\delta_{5\to7}\delta_{6\to7}.$$

If we now isolate $\delta_{4\to7}$ we get

$$\delta_{4\to7} = \frac{\sum_b \delta_{1\to4}\delta_{2\to4}\delta_{5\to7}\delta_{6\to7}}{\delta_{5\to7}\delta_{6\to7}}.$$

After we cancel out the common terms $\delta_{5\to7}$ and $\delta_{6\to7}$, we get the desired form.

This message is essentially identical to the message in a cluster graph where we marginalize the other incoming messages in the cluster to send a message to a particular sepset. Here region 4 behaves as a cluster and region 7 as a sepset. The other messages incoming to region 7 have a similar form.

Messages into the middle layer regions have more complex form. For example

$$\delta_{1\to4} = \frac{\sum_A \psi_1 \delta_{3\to5}}{\delta_{5\to7}}.$$

Again, we can use the marginal consistency constraint

$$\beta_4 = \sum_A \beta_1$$

to reconstruct the message. Plugging in the expanded form of the two beliefs, and isolating $\delta_{1\to4}$ we get

$$\delta_{1\to4} = \frac{\sum_A \psi_1 \delta_{2\to4}\delta_{3\to5}\delta_{6\to7}}{\delta_{2\to4}\delta_{5\to7}\delta_{6\to7}}.$$

After we cancel out $\delta_{2\to4}$ and $\delta_{6\to7}$, we get the desired form. ■

These definitions set up a message passing algorithm similar to CGraph-SP-Calibrate, except that we use the messages as formulated in equation (11.37). As with belief propagation on cluster graphs, we can prove that convergence points of such propagations are stationary points of the RegionGraph-Optimize optimization problem.

Theorem 11.7 *A set of beliefs $\boldsymbol{Q}$ is a stationary point of* RegionGraph-Optimize *for region graph $\mathcal{R}$ if and only if for every edge $(i\text{–}j) \in \mathcal{E}_{\mathcal{R}}$ there are auxiliary factors $\delta_{u \to d}(\boldsymbol{C}_d)$ that satisfy equation (11.36) and equation (11.37).*

This result is a direct generalization of theorem 11.5, and is proved in a similar way. We leave the detail as an exercise (see exercise 11.14). Much of the discussion following theorem 11.5 applies here. In particular, we do not have guarantees that iterations of message passing will converge. However, if they do, we have reached a stationary point of the energy functional. In practice, the experience is that when we consider moving from the Bethe approximation to "richer" region graphs that contain intermediate regions with larger subsets, problems of nonconverging runs are less common. For example, a region graph construction for grids is much more convergent than the corresponding cluster graph (see exercise 11.15). However, except for special cases (for example, region graphs that correspond to cluster trees), we do not know how to characterize region graphs where belief propagation converges.

11.3.8 Discussion

Cluster-graph belief propagation methods such as the ones we have described in this chapter provide a general-purpose mechanism for approximate inference in graphical models. In principle, they apply to any network, including networks with high tree-width, for which exact inference is intractable. They have been applied successfully to a large number of dramatically different applications, including (among many others) message decoding in communication over a noisy channel (see box 11.A), predicting protein structure (see box 20.B), and image segmentation (see box 4.B).

However, **it is important to keep in mind that cluster-graph belief propagation is not a global panacea to the problem of inference in graphical models. The algorithm may not converge, and when it does converge, there may be multiple different convergence points.** Although there are currently no conditions characterizing precisely when cluster-graph belief propagation converges, several factors seem to play a role.

The first is the topology of the network: A network containing a large number of short loops is more likely to be nonconvergent. Although this notion has been elusive to characterize in practice, it has been shown that cluster-graph belief propagation is guaranteed to converge on networks with a single loop.

An even more significant factor is the extent to which the factors parameterizing the network are skewed, or close to deterministic. Intuitively, deterministic factors can cause difficulties in several ways. First, they often induce strong correlations between variables, which cluster-graph belief propagation (depending on the approximation chosen) can lose. This error can have an effect not only for the correlated variables, but also for marginals of variables that interact with both. Second, close-to-deterministic factors allow information to be propagated reliably through long paths in the network. Recall that part of our motivation for the running intersection property was to prevent information about some variable to be propagated infinitely through a

loop. While the running intersection property prevents such loops from occurring structurally, deterministic potentials allow us to recreate them using an appropriate choice of parameters. For example, if A is deterministically equal to B, then we can have a cycle of clusters where A appears in some of the clusters and B in others. Although this cluster graph may satisfy the running intersection property relative to A, effectively there is a cycle in which the same variable appears in all clusters. Finally, as we discussed in section 11.3.4, factors that are less skewed provide smoothing of the messages, reducing oscillations; indeed, one can even prove that, if the skew of the factors in the network is sufficiently bounded, it can give rise to a contraction property that guarantees convergence.

☞ **In summary, the key factor relating to convergence of belief propagation appears to be the extent to which the network contains strong influences that "pull" a variable in different directions. Owing to its local nature, the algorithm is incapable of reconciling these different constraints, and it can therefore oscillate as different messages arrive that pull it in one direction or another.**

A second problem relates to the quality of the results obtained. Despite the appeal and importance of the energy-based analysis, it does not (except in a few rare cases — see section 11.7) provide any guarantees on the accuracy of the marginals obtained by cluster-graph belief propagation. This is in contrast to the sampling-based methods of chapter 12, where we are at least assured that, if we run the algorithm for long enough, we will obtain accurate estimates of the posteriors. (Of course, the key question of "how long is long enough" does not usually have an answer, so it is not clear how important this distinction is in practice.) Empirical results show that, in the settings where cluster-graph belief propagation convergence is more likely (not too many tight loops, no highly skewed factors), one also often obtains reasonable answers.

Importantly, these answers are often good but overconfident: The value $x \in Val(X)$ to which cluster-graph belief propagation gives the highest probability is often the value for which $P_\Phi(X = x)$ is indeed the highest, but the probability assigned to x by the approximation is often too high. This phenomenon arises (partly) from the fact the cluster-graph belief propagation ignores correlations between messages and can therefore count the same piece of evidence multiple times as it arrives along different paths, leading to overly strong conclusions. In other cases, however, the answers obtained by cluster-graph belief propagation are simply wrong (see section 11.3.1); unfortunately, there is currently no way of determining when the answers returned by a run of cluster-graph belief propagation are reasonable approximations to the true marginals.

The intuitions described previously do help us, however, to design approximations that are more likely to produce good answers. In general, we cannot construct a cluster graph that preserves all of the higher-order interactions among the factors. Hence, we need to decide which factors to include in the cluster graph and how to relate them. As the preceding discussion suggests, we do better if we construct approximations that incorporate tight loops and maintain the strongest factors within clusters as much as possible. While these intuitions provide reasonable rules of thumb on how to construct approximations, it is not obvious how to capture them within a general-purpose automated cluster-graph construction procedure.

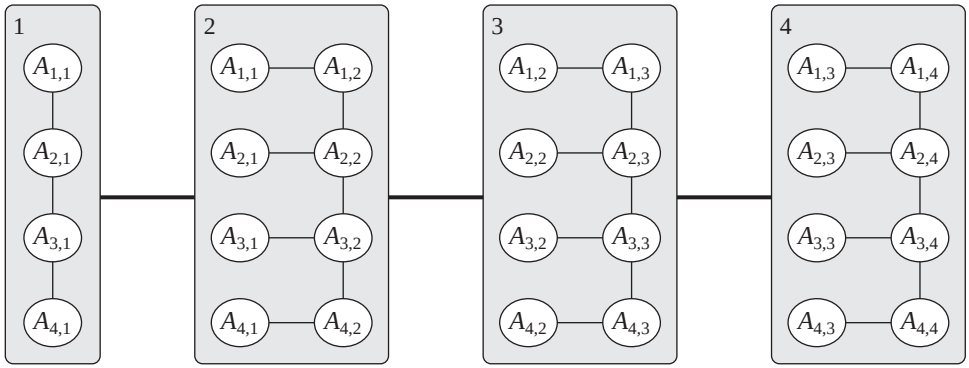


Figure 11.12 A cluster for a 4×4 grid network. The structure within each cluster represents the arcs whose factors are assigned to that cluster.

11.4 Propagation with Approximate Messages ⋆

The cluster-graph belief propagation methods achieved approximation by "relaxing" the requirement of having a cluster tree. Instead, we used a cluster graph and constructed an approximation by a set of pseudo-marginals. This approximation avoided the need to construct large clusters, which incur an exponential cost in terms of memory and running time. **In this section, we consider an approach in which the simplification occurs within a given cluster structure; rather than simplify this structure, we perform approximate propagation steps within it. This allows us to keep the correct clusters and gain efficiency by using more compact representations and operations on these clusters (at the cost of introducing approximations).** Importantly, this approach is orthogonal to the methods we described in the previous section, in that approximate message passing can occur both within a clique tree or a cluster graph approximation. For ease of presentation, we focus on the case of clique trees in this section, but the ideas easily carry through to the more general setting.

The basic concept behind the methods described in this section is the use of approximate messages in clique tree (or cluster graph) propagation. Instead of representing messages in the clique trees as factors, we use more compact representations of approximate factors. There are many different schemes that we can use for approximating messages. To ground the discussion, we begin in section 11.4.1 by describing one important instantiation of this general framework, which is very natural in our setting — message approximation using a factored form (for example, as a product of marginals). In section 11.4.2 and section 11.4.3 we then discuss algorithms that perform message passing using approximate messages. In section 11.4.4 we describe a general algorithm, called *expectation propagation*, that applies to any approximation in the exponential family. Finally, in section 11.4.5, we show that the expectation propagation algorithm for the exponential family can be derived from a variational analysis, similar to the ones we discussed in the previous section.

expectation propagation

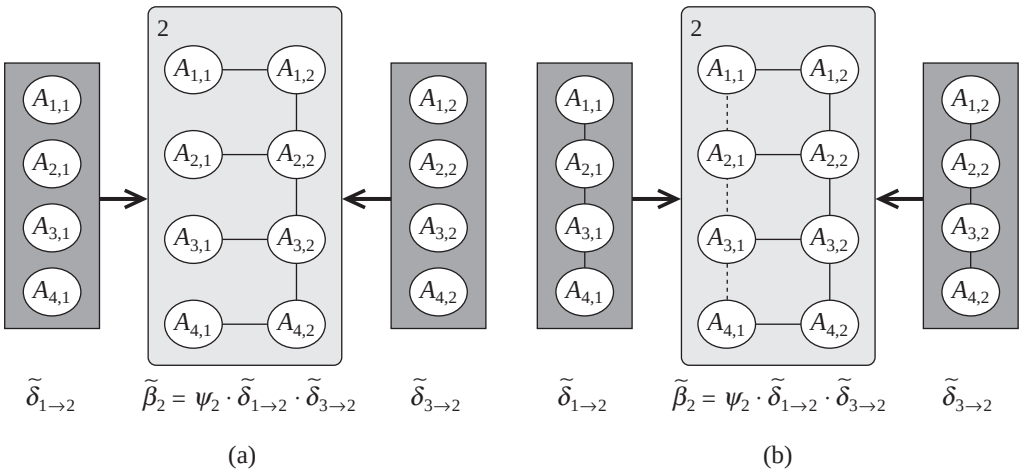


Figure 11.13 Effect of different message factorizations on the structure of the belief factor $\tilde{\beta}_2$ in the example of figure 11.12.

11.4.1 Factorized Messages

We begin by considering a concrete example where message approximation may be useful. Consider again the 4×4 grid network of figure 11.4. We can construct a cluster tree for this network as shown in figure 11.12. (Note that this cluster tree is not the optimal one for this network.) In our discussion of inference until now, we ignored the inner structure of clusters and treated the cluster as being associated with a single factor. Now, however, we keep track of the structure of the original factors associated with the cluster. We will see shortly why keeping this structure will help us. In figure 11.12 this structure is portrayed by "subnetworks" within each cluster.

Calibration in this cluster tree involves sending messages that are joint measures over four variables. An intuitive idea is to simplify the messages by using a factored representation. For example, consider the message $\delta_{1\to 2}$, and suppose that we approximate it by a factored form

$$\tilde{\delta}_{1\to 2}[A_{1,1}, A_{2,1}, A_{3,1}, A_{4,1}] = \tilde{\delta}_{1\to 2}[A_{1,1}]\tilde{\delta}_{1\to 2}[A_{2,1}]\tilde{\delta}_{1\to 2}[A_{3,1}]\tilde{\delta}_{1\to 2}[A_{4,1}].$$

What can we gain from such an approximation? Clearly, this form provides a more concise representation of the message. Instead of representing the message with values for the sixteen possible joint assignments, we send a message with two values for each variables (leading to a total of eight "parameters"). If we consider bigger grids, then the saving will be more substantial.

Does this saving gain us efficiency? Naively, the answer is no. Recall that the main computational cost of exact inference is generating the message in a cluster. This requires multiplying incoming messages with the original factor of the cluster and marginalization. In our example, $\boldsymbol{C}_2$ involves eight variables, and so this operation should involve operations over the $2^8 = 256$ joint values of these variables. Thus, one might argue that the saving of eight parameters in representing the message does not deal with the core computational problem we have at hand

and leads to negligible improvement.

It turns out, however, that if we consider the internal structure of the cluster we can exploit the factored form of the message. Consider computing the beliefs over $\boldsymbol{C}_2$ given messages from $\boldsymbol{C}_1$ and $\boldsymbol{C}_3$. In exact computation, we multiply the potential ψ_2 with $\delta_{1\to 2}$ and $\delta_{3\to 2}$ and normalize to get the beliefs about $\boldsymbol{C}_2$. However, if we approximate both messages by a product of univariate terms, we notice that the product of the messages with the factors in $\boldsymbol{C}_2$ forms a network structure that we can exploit. In our example, this network is a tree-structured network shown in figure 11.13a. We can easily calibrate this network and answer queries about the beliefs over $\boldsymbol{C}_2$ without enumerating all joint assignments to variables in this cluster.

We can get similar savings even if we use a richer approximation that can better capture dependencies among variables in the message. Suppose we approximate $\delta_{1\to 2}$ by a factored form that corresponds to the chain structure $A_{1,1}$—$A_{2,1}$—$A_{3,1}$—$A_{4,1}$. This approximation makes conditional independence assumptions about the variables, but it captures some of the dependencies among them. In this example, this representation actually captures the message $\delta_{1\to 2}$. However, we can check that $\delta_{3\to 2}$ does not satisfy the conditional independence of $A_{1,2}$ and $A_{3,2}$ given $A_{2,2}$. Thus, in this case, a chain representation is an approximation. If we multiply these two approximations with the cluster $\boldsymbol{C}_2$, we get a set of factors that has the structure shown in figure 11.13b. Although not a tree, this graph has a tree-width of 2, regardless of the grid size. Thus, once again, we can use exact inference methods on the resulting product of factors.

We can exploit this intuition by maintaining both the initial potentials and the messages as *factor sets*. For the initial potential, these factors are the parameterization of the original network; for messages, these factors are introduced by the approximation. A factor set $\vec{\phi} = \{\phi_1, \ldots, \phi_k\}$ provides a compact representation for the higher-dimensional factor $\phi_1 \cdot \phi_2 \cdot \cdots \cdot \phi_k$.

factor set

Recall that belief propagation involves two main operations: product and marginalization. The *product* of factor sets is easy. Suppose we have the factor sets $\vec{\phi}_1$ and $\vec{\phi}_2$. The factor set $\vec{\phi}_1 \cdot \vec{\phi}_2$ is simply the factor set that contains the union of the two factor sets (we assume that the components of the factor sets are distinct).

factor set product

The difficult operation is *marginalization*. Suppose we have a factor set $\vec{\phi} = \{\phi_1, \ldots, \phi_k\}$, and we consider the marginalization $\sum_X \vec{\phi}$. This operation couples all the factors that contain X. In general, for a well-constructed clique tree, the message that results from marginalizing a clique will not satisfy any conditional independence statements and therefore cannot be factorized (see exercise 11.21).

factor set marginalization

Returning to our example of figure 11.12, one of our inference steps is to compute:

$$\delta_{2\to 3} = \sum_{A_{1,1}, A_{2,1}, A_{3,1}, A_{4,1}} \psi_2 \cdot \delta_{1\to 2}.$$

To achieve the efficient inference steps we discussed earlier, we want to approximate $\delta_{2\to 3}$ by a product of simpler terms. This is an instance of a problem we encountered in section 8.5 — approximating a given distribution by another distribution from a given family of distributions. In our case, the approximating family is the set of distributions that take a particular product form. As we discussed in section 8.5, there are several ways of projecting a distribution P into some constrained class of distributions $\mathcal{Q}$. Indeed, throughout our discussion in this chapter so far, we have been searching for a distribution Q which is the I-projection of P_Φ — the one

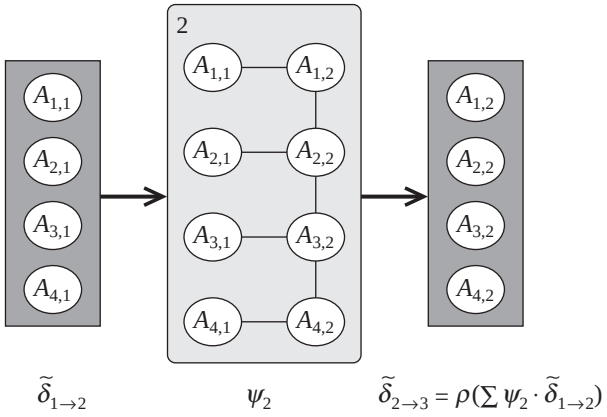


Figure 11.14 Example of propagation in cluster tree with factorized messages

M-projection

that minimizes $\boldsymbol{D}(Q\|P_\Phi)$. However, as we discussed in section 8.5, we cannot compute the I-projection of a distribution in closed form. Conversely, the *M-projection* for many classes $\mathcal{Q}$ of factored distributions has an easy closed form expression. For example, the M-projection of P onto the class of factored distributions is simply the product of marginals of P (see proposition 8.3). We can compute these marginals by computing the marginals of the individual variables in the message. We note that the message is often not normalized, and is therefore not a distribution; however, we can normalize the message and treat it as a distribution without changing the final posterior obtained by the inference algorithm (see exercise 9.3).

Given the simplicity of M-projections, one might wonder why we used I-projections in our discussion so far. There are two main reasons. First, the theory of energy functionals allows us to use I-projections to lower-bound the partition function. Second, and more importantly, computing marginal probabilities for our distribution P_Φ is generally intractable. Thus, although the M-projection has a simple form, it is infeasible to apply when we have an intractable network structure. In our current setting, the situation is different. Here, our projection task is to approximate the outgoing message from a cluster $\boldsymbol{C}$. As discussed earlier, we assume that the product of the beliefs and the approximated messages for $\boldsymbol{C}$ is a factor set with tractable structure. This assumption allows us to use exact inference *within the cluster* $\boldsymbol{C}$ to compute the marginal probabilities needed for M-projections.

11.4.2 Approximate Message Computation

We now discuss in greater detail the task of computing factorized messages. To begin, consider the projection of $\delta_{2\to 3}$ into a fully factorized representation. According to proposition 8.3, to build the M-projection of $\delta_{2\to 3}$ onto a factored distribution, we need to compute the marginals of $A_{1,2}$, $A_{2,2}$, $A_{3,2}$, and $A_{4,2}$ in $\delta_{2\to 3}$. Since $\delta_{2\to 3}$ is defined to be the marginal of the factor set $\psi_2 \cdot \delta_{1\to 2}$, we need to compute terms such as $P_{\psi_2\cdot\delta_{1\to 2}}(A_{1,2})$, where we use $P_{\psi_2\cdot\delta_{1\to 2}}$ to denote the measure represented by the factor set. At an abstract level, this operation involves computing $\delta_{2\to 3}$ and then projecting it onto the simpler representation. However, if we examine

the structure of this factor set (see figure 11.14), we see that this computation can be done using standard exact inference on the factor set. In this particular case, the factor set is a tree network, and so inference is cheap. Similarly, we can compute the marginals over the variables in $\delta_{2\to 3}$. Thus, we can use the properties of M-projections and exact inference to compute the resulting projected message without ever explicitly enumerating the joint over the four variables in $\delta_{2\to 3}$. In an $n \times n$ grid, for example, we can perform these operations in time that is linear in n, whereas the explicit computation would have constructed a factor of exponential size. As we discussed, the more complex approximation of figure 11.13b also has a bounded tree-width of 2, and therefore also allows linear time inference.

Algorithm 11.4 Projecting a factor set to produce a set of marginals over a given set of scopes

```
Procedure Factored-Project (
    φ⃗,    // an input factor set
    𝒴,    // A set of desired output scopes
)
1   Build an inference data structure 𝒰
2   Initialize 𝒰 with factors in φ⃗
3   Perform inference in 𝒰
4   φ⃗^o ← ∅
5   for all Y_j ∈ 𝒴
6     Find a clique C_j in 𝒰 so that Y_j ⊆ C_j
7     ψ_j ← β_𝒰(Y_j)    // marginal of Y_j in 𝒰
8     φ⃗^o ← φ⃗^o ∪ {ψ_j}
9   return φ⃗^o
```

The overall structure of the algorithm is shown in algorithm 11.4. At a high level, we can view exact inference in the factor set as a "black box" and not concern ourselves with the exact implementation. However, it is also useful to consider how this type of projected message may be computed efficiently. Most often, the inference data structure $\mathcal{U}$ is a cluster graph or a clique tree, into which the initial factors $\vec{\phi}$ can easily be incorporated, and from which the target factors, of the appropriate scopes, can be easily extracted. To allow for that, we typically design the cluster graph to be family-preserving with respect to both sets of factors. Under that assumption, we can extract a factor ψ_j over the scope $\boldsymbol{Y}_j$ by identifying a cluster $\boldsymbol{C}_j$ in $\mathcal{U}$ whose scope contains $\boldsymbol{Y}_j$, and marginalizing it over $\boldsymbol{Y}_j$. As an alternative approach, we can construct an unconstrained clique tree, and use the out-of-clique inference algorithm of section 10.3.3.2 to extract from the graph the joint marginals of subsets of variables that are not together in a clique. (We note that out-of-clique inference is more challenging in the context of cluster graphs, since the path used to relate the clusters containing the query variables can affect the outcome of the computation; see exercise 11.22.)

If our representation of a message is simply a product of marginals over disjoint subsets of variables, this algorithm suffices to produce the output message: We produce a factor over each (disjoint) scope $\boldsymbol{Y}_j$; the product of these factors is the M-projection of the distribution. But for

richer approximations, the required operation is somewhat more complex.

Example 11.6

Consider again our grid example of figure 11.13b. Here, the clique needs to take in messages that involve factors over pairs of variables $A_{i,1}, A_{i+1,1}$ and produce messages over pairs of variables $A_{i,2}, A_{i+1,2}$. We can accomplish this goal by constructing, within this clique, a nested clique tree that has at least one clique containing each of these pairs. For this purpose, we can use any clique tree based on triangulating the structure inside the clique in figure 11.13b. For example, we can have a tree with the cliques $\{A_{1,1}, A_{1,2}, A_{2,2}\}$, $\{A_{1,1}, A_{2,1}, A_{2,2}\}$, $\{A_{2,1}, A_{2,2}, A_{3,2}\}$, $\{A_{2,1}, A_{3,1}, A_{3,2}\}$, $\{A_{3,1}, A_{3,2}, A_{4,2}\}$, $\{A_{3,1}, A_{4,1}, A_{4,2}\}$.

Our goal now is to produce a set of factors that is the M-projection of the true message onto the chain. Assume that we extract from the clique tree the pairwise marginals $P(A_{1,2}, A_{2,2})$, $P(A_{2,2}, A_{3,2})$, and $P(A_{3,2}, A_{4,2})$. However, we cannot directly encode the distribution using these factors as a factor set, since this approach would double-count the probability of the singletons $A_{2,2}$ and $A_{3,2}$, each of which appears in two factors. To produce the true M-projected distribution, we need to divide by the double-counted marginals $A_{2,2}$ and $A_{3,2}$.

We can achieve this correction easily by computing these node marginals and adding to our factor set representation two factors

$$\frac{1}{\tilde{\delta}_{2\to 3}(A_{2,2})}, \frac{1}{\tilde{\delta}_{2\to 3}(A_{3,2})}$$

that compensate for the double-counting. This factor set represents the M-projection of the distribution, and it can be used in subsequent message passing steps. Equivalently, we are representing the distribution as a calibrated clique tree over $A_{1,2}, A_{2,2}, A_{3,2}, A_{4,2}$, where the factors derived from the pairwise marginals are the clique beliefs, the factors derived from inverted node marginals are the sepset messages, and the overall distribution is encoded as in equation (10.11). ■

We can easily generalize this approach to more complex message representations. Most generally, assume that we choose to encode our message between cluster i and cluster j using a representation as in equation (11.35); more precisely, we have some set of factors $\{\beta_r(\boldsymbol{X}_r) : \boldsymbol{X}_r \in \mathbf{R}_{i,j}\}$, each raised to some power:

$$\tilde{\delta}_{i\to j} = \prod_{r\in\mathbf{R}_{i,j}} (\beta_r(\boldsymbol{X}_r))^{\kappa_r}. \tag{11.38}$$

We can compute this approximation in our procedure using the Factored-Project algorithm, using the set $\{\boldsymbol{X}_r \in \mathbf{R}_{i,j}\}$ as $\mathcal{Y}$. The output of this procedure is a set of calibrated marginals over these scopes, and can therefore be plugged into equation (11.38) to produce a message in the appropriate factored form. The same factors, each raised to its appropriate power, can be used as the factorized message passed to cluster j.

Importantly, we note that this approach does not always provide an exact M-projection of the true message. This property is guaranteed only when the set of regions forms a clique tree, allowing the M-projection to be calculated in closed form using equation (11.38); in other cases, we obtain only an approximation to the M-projection. In practice, we often choose some simple clique tree, as in example 11.6, to encode our distribution, allowing the M-projection to be performed easily and exactly. However, in some cases, a clique tree approximation, to stay tractable, is forced to make too many independence assumptions, leading to a poor

approximation of the message. Hence, the approach of an approximation M-projection into a richer class of distributions may provide a useful alternative in some cases.

In summary, we have shown how we can perform an approximate message passing step with structured messages. The algorithm maintains messages and beliefs in factored form. Factors are entered into a nested clique tree or cluster graph, and message propagation is used to compute the factors describing the output message. In the fully factored case, this representation is simply a set of factors, and the multiplication operation used in message passing is simply a union of factor sets. In the more complex setting, we may need to postprocess the set of marginals — exponentiating them by appropriate counting numbers — to eliminate double-counting. The resulting set of factors is the compact representation of the outgoing message.

11.4.3 Inference with Approximate Messages

Equipped with these operations on factor sets, we can consider how to use these tools within cluster tree propagation. Our algorithm will maintain the beliefs at each cluster i using a factor set $\vec{\phi}_i$. Initially, we are given a cluster tree $\mathcal{T}$ with an assignment of the original factors to clusters. We have also determined the factorized form for each sepset, in terms of a network structure $\mathcal{G}_{i,j}$ that describes the desired factorization for $\tilde{\delta}_{i \to j}$ and $\tilde{\delta}_{j \to i}$.

There are two main strategies for applying the ideas described in the previous section to define an approximate message passing algorithms in clique trees. One is based on a sum-product message passing scheme, and the other on belief update messages. As we will see, although these two strategies are equivalent in the exact inference setting, they lead to fairly different algorithms when we consider approximations.

For notational simplicity, we introduce the notation M-project-distr$_{i,j}$ to be the combined operation of marginalizing variables that do not appear in $\boldsymbol{S}_{i,j}$ and performing the M-projection onto the set of distributions that can be factorized according to $\mathcal{G}_{i,j}$.

11.4.3.1 Sum-Product Propagation

Consider the application of sum-product propagation (CTree-SP-Calibrate, algorithm 10.2), to our grid example. In this case, we need to perform the following operations:

$$\begin{aligned}
\delta_{1 \to 2} &= \psi_1 \\
\delta_{2 \to 3} &= \sum_{A_{1,1}, \ldots, A_{4,1}} \psi_2 \cdot \delta_{1 \to 2} \\
\delta_{3 \to 4} &= \sum_{A_{1,2}, \ldots, A_{4,2}} \psi_3 \cdot \delta_{2 \to 3} \\
\delta_{4 \to 3} &= \sum_{A_{1,4}, \ldots, A_{4,4}} \psi_4 \\
\delta_{3 \to 2} &= \sum_{A_{1,3}, \ldots, A_{4,3}} \psi_3 \cdot \delta_{4 \to 3} \\
\delta_{2 \to 1} &= \sum_{A_{1,2}, \ldots, A_{4,2}} \psi_2 \cdot \delta_{3 \to 2}.
\end{aligned}$$

A straightforward application of approximation is to replace each of the messages by the M-projected version:

$$\begin{aligned}
\tilde{\tilde{\delta}}_{1\to 2} &= \text{M-project-distr}_{1,2}(\psi_1) \\
\tilde{\tilde{\delta}}_{2\to 3} &= \text{M-project-distr}_{2,3}(\psi_2 \cdot \tilde{\tilde{\delta}}_{1\to 2}) \\
\tilde{\tilde{\delta}}_{3\to 4} &= \text{M-project-distr}_{3,4}(\psi_3 \cdot \tilde{\tilde{\delta}}_{2\to 3}) \\
\tilde{\tilde{\delta}}_{4\to 3} &= \text{M-project-distr}_{3,4}(\psi_4) \\
\tilde{\tilde{\delta}}_{3\to 2} &= \text{M-project-distr}_{2,3}(\psi_3 \cdot \tilde{\tilde{\delta}}_{4\to 3}) \\
\tilde{\tilde{\delta}}_{2\to 1} &= \text{M-project-distr}_{1,2}(\psi_2 \cdot \tilde{\tilde{\delta}}_{3\to 2}).
\end{aligned}$$

Each of these projection operations can be performed using the procedure described in the previous section.

sum-product expectation propagation

More generally, our *sum-product expectation propagation* procedure is identical to sum-product propagation of section 10.2, except that we modify the basic propagation procedure SP-Message so that, rather than simply marginalizing the product of factors, it computes their M-projection. Otherwise, the general structure of the propagation procedure is maintained exactly as before. Each cluster collects the messages from its neighbors and when possible sends outgoing messages. As for the original sum-product message passing, this process converges after a single upward-and-downward pass over the clique tree structure. Thus, unlike most of the other approximations we discuss in this chapter, this procedure terminates in a fixed number of steps.

Note that, after performing the propagation, the final beliefs over the individual clusters in the tree are not an explicit representation of a joint distribution over the variables in the cluster. In this case, the final beliefs over a cluster $\boldsymbol{C}$ are represented in a factorized form, as a set of beliefs. In our running example, the computed beliefs over $\boldsymbol{C}_2$ have the form

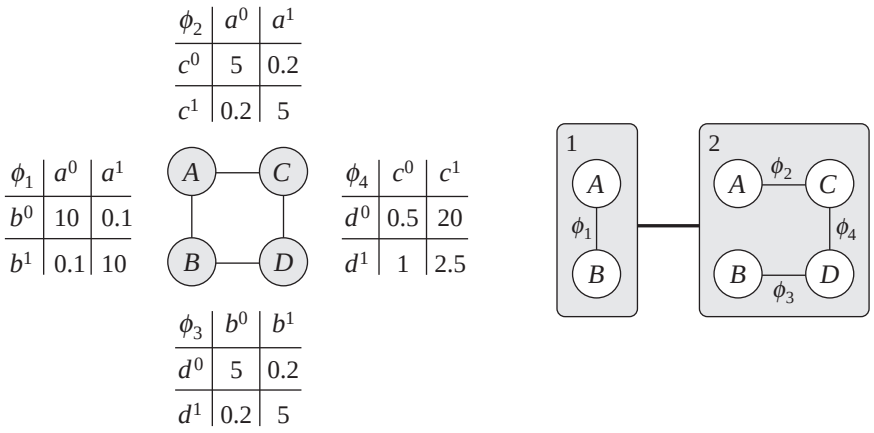
$$\tilde{\beta}_2 = \psi_2 \cdot \tilde{\tilde{\delta}}_{1\to 2}\tilde{\tilde{\delta}}_{3\to 2},$$

and the structure shown in figure 11.13a. Because this structure allows tractable inference, we can answer queries about the posterior distribution of these variables using standard inference methods, such as variable elimination or clique tree inference.

These computational benefits come at a price. The exact beliefs over $\boldsymbol{C}_2$ are not decomposable (see exercise 11.20). And thus, although forcing a particular independence structure on the marginal distribution has computational advantages, it does lose information. With this caveat in mind, we can still question whether the algorithm finds the best possible approximation within the set of constraints we are considering.

Example 11.7

To get a sense of the issues, consider the simple example shown in figure 11.15a. In this small network, the potential $\phi_1(A, B)$ introduces a strong coupling between A and B, with a strong preference for $A = B$. The potentials $\phi_2(A, C)$, $\phi_3(B, D)$ incur weaker coupling between A and C and B and D. Finally, the potential $\phi_4(C, D)$ has a strong preference for $C = c^1$, with a weaker preference for $C \neq D$.



(a) Markov network (b) cluster tree

Figure 11.15 Markov network used to demonstrate approximate message passing. (a) A simple Markov network with four pairwise potentials. (b) A clique tree for this network.

If we perform exact inference in this network, we find the following marginal posteriors:

$$\begin{array}{lcllcl} P(a^0, b^0) & = & 0.274 & P(c^0, d^0) & = & 0.102 \\ P(a^0, b^1) & = & 0.002 & P(c^0, d^1) & = & 0.018 \\ P(a^1, b^0) & = & 0.041 & P(c^1, d^0) & = & 0.368 \\ P(a^1, b^1) & = & 0.682 & P(c^1, d^1) & = & 0.512. \end{array}$$

We see that the preference for $C = c^1$ is reflected in this distribution (with the marginal distribution $P(c^1) = 0.88$). In addition, the strong coupling between A and B is propagated through the network, which results in making $D = d^1$ more probable when $C = c^1$.

What happens when we perform inference using the cluster tree of figure 11.15b and use approximate messages that are products of marginals? It is easy to see that, because ϕ_1 is symmetric, we get that $\tilde{\delta}_{1\to 2}[a^1] = 0.5$, and $\tilde{\delta}_{1\to 2}[b^1] = 0.5$. We can compare the exact message and the approximate one

$$\begin{array}{lcllcl} \delta_{1\to 2}(a^0, b^0) & = & 0.495 & \tilde{\delta}_{1\to 2}(a^0, b^0) & = & 0.5 * 0.5 = 0.25 \\ \delta_{1\to 2}(a^0, b^1) & = & 0.005 & \tilde{\delta}_{1\to 2}(a^0, b^1) & = & 0.5 * 0.5 = 0.25 \\ \delta_{1\to 2}(a^1, b^0) & = & 0.005 & \tilde{\delta}_{1\to 2}(a^1, b^0) & = & 0.5 * 0.5 = 0.25 \\ \delta_{1\to 2}(a^1, b^1) & = & 0.495 & \tilde{\delta}_{1\to 2}(a^1, b^1) & = & 0.5 * 0.5 = 0.25. \end{array}$$

Thus, the approximate message is one where each joint assignment to A and B is equiprobable. This approximation loses the coupling that ϕ_1 introduces between A and B, and therefore it is a poor approximation to the exact message.

Next, we multiply this approximate message into the clique $\boldsymbol{C}_2$. The initial factor here is $\psi_2 =$

$\phi_2 \cdot \phi_3 \cdot \phi_4$, and after multiplying it with $\tilde{\delta}_{1\to 2}$ we get the beliefs

$$\begin{aligned}\tilde{\beta}_2(c^0, d^0) &= 0.021\\ \tilde{\beta}_2(c^0, d^1) &= 0.042\\ \tilde{\beta}_2(c^1, d^0) &= 0.833\\ \tilde{\beta}_2(c^1, d^1) &= 0.104.\end{aligned}$$

Note that this factor is essentially a normalization of ϕ_4, since the message $\tilde{\delta}_{1\to 2}$ puts a uniform distribution of A and B, and since ϕ_2 and ϕ_3 are symmetric. Because the information about the coupling between A and B is not propagated into this cluster, we lose the consequent coupling between C and D, and the resulting approximation to $P(C, D)$ is also quite poor.

By contrast, if we now compute $\tilde{\delta}_{2\to 1}$, we get that

$$\begin{aligned}\tilde{\delta}_{2\to 1}[a^1] &= 0.904\\ \tilde{\delta}_{2\to 1}[b^1] &= 0.173.\end{aligned}$$

This message ascribes high probability to a^1 and a low one to b^1. This is quite different from the original coupling introduced by ϕ_1. Thus, when we combine the two to get the approximated posterior over A and B, we get the following beliefs factor:

$$\begin{aligned}\tilde{\beta}_1(a^0, b^0) &= 0.326\\ \tilde{\beta}_1(a^0, b^1) &= 0.001\\ \tilde{\beta}_1(a^1, b^0) &= 0.031\\ \tilde{\beta}_1(a^1, b^1) &= 0.642.\end{aligned}$$

This approximation is fairly close to the exact marginal over A and B. ■

The problem in this example is that the approximation of $\tilde{\delta}_{1\to 2}$ is done blindly, not taking into account its effect on approximations on computations in downstream clusters. Thus, the message did not place sufficient emphasis on obtaining a correct approximation for the case $A = a^1$, which roughly corresponds to the "important" (high-probability) case $C = c^1$.

11.4.3.2 Belief Update Propagation

Example 11.7 shows a case where factorizing messages leads to a big error in the approximated beliefs. Let us examine this example in somewhat more detail. Given the update of $\tilde{\delta}_{2\to 1}$, the approximation of $P(A, B)$ is more or less on target. Can we use this information to improve our approximation of $P(C, D)$? To do this, we would need to revise $\tilde{\delta}_{1\to 2}$. The posterior over A and B informs us that most of the mass of the probability distribution is on a^1, b^1. With this information, we might want to change $\tilde{\delta}_{1\to 2}$ to reflect preferences for a^1 and b^1.

A priori, it appears that this idea is inherently problematic; after all, a key constraint for exact inference is to avoid feedback from $\delta_{2\to 1}$ to $\delta_{1\to 2}$, so as not to double-count evidence. For this reason, we took care, in the sum-product message-passing algorithm, *not* to multiply in the message $\delta_{2\to 1}$ when passing messages from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$.

However, recall that in section 10.3, we presented the sum-product-divide update rule and showed its equivalence to the sum-product rule. Briefly recapping, we can take the sum-product

update rule:

$$\delta_{i \to j} = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \left(\prod_{k \in \text{Nb}_i - \{j\}} \delta_{k \to i}, \right)$$

and multiply and divide by $\delta_{j \to i}$, resulting in the rule:

$$\delta_{i \to j} \leftarrow \frac{\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i}{\delta_{j \to i}}.$$

These two rules are therefore equivalent *in the exact case*. However, when we consider approximate inference, the situation is more complex.

belief-update expectation propagation

Consider now performing *belief-update expectation propagation* message passing. Assume, as before, that the algorithm is maintaining a set of approximate beliefs $\tilde{\beta}_i$. We now have two possible stages in which to do the project. The first is:

$$\tilde{\delta}_{i \to j} \leftarrow \text{M-project-distr}_{i,j} \left(\frac{\tilde{\beta}_i}{\tilde{\delta}_{j \to i}} \right).$$

This version is identical to the approximate sum-product we discussed before: The beliefs factor β_i accounts for all of the incoming messages; when we divide by the message $\tilde{\delta}_{j \to i}$ before projecting, we are projecting the product of all the other incoming message, which is precisely the sum-product message.

Alternatively, we can do the projection before we divide by $\tilde{\delta}_{j \to i}$. In this second approach, we first project $\tilde{\beta}_i$, and then divide by $\tilde{\delta}_{j \to i}$:

$$\begin{aligned} \tilde{\sigma}_{i \to j} &\leftarrow \text{M-project-distr}_{i,j}(\tilde{\beta}_i) \\ \tilde{\delta}_{i \to j} &\leftarrow \frac{\tilde{\sigma}_{i \to j}}{\tilde{\delta}_{j \to i}}. \end{aligned} \tag{11.39}$$

In this update, we first collect all messages into $\boldsymbol{C}_i$; we then compute the beliefs about $\boldsymbol{C}_i$ and project this to the required form of the message. As in the exact belief update algorithm, this term accounts for information sent from $\boldsymbol{C}_j$. Note that both $\tilde{\sigma}_{i \to j}$ and $\tilde{\delta}_{j \to i}$ have the same factorization, and hence so does their quotient $\tilde{\delta}_{i \to j}$.

This message, which is subsequently used to update $\boldsymbol{C}_j$, is very different from the sum-product update. This is because the incoming message was used in determining the approximation. 

The approximation process is invariably a trade-off, in that a better approximation of some regions of the probability space results in a worse approximation in others. In the belief-update form of message passing, we can take into account the current approximation of the message from the target clique when deciding on our approximation, potentially focusing more of our attention on "more relevant" parts of the space.

To integrate this update rule into the standard belief-update message passing algorithm, we simply replace BU-Message of algorithm 10.3 with EP-Message, shown in algorithm 11.5. We note that the data structures in this procedure are slightly different from those in the original algorithm. First, we maintain the cluster beliefs implicitly, as a factor set, and consider the product of these factors only as part of the M-projection operation. Second, we do not only

Algorithm 11.5 Modified version of BU-Message **that incorporates message projection**

Procedure EP-Message (
  i, // sending clique
  j // receiving clique
)
1 $\tilde{\sigma}_{i \to j} \leftarrow$ M-project-distr$_{\mathcal{Q}_{i,j}}(\vec{\phi}_i)$
2 // marginalize and project the clique over the sepset
3 Remove old $\tilde{\delta}_{i \to j}$ from $\vec{\phi}_j$
4 $\tilde{\delta}_{i \to j} \leftarrow \frac{\tilde{\sigma}_{i \to j}}{\tilde{\delta}_{j \to i}}$
5 // divide by the message from from $\boldsymbol{C}_j$ to $\boldsymbol{C}_i$
6 Insert new $\tilde{\delta}_{i \to j}$ into $\vec{\phi}_j$

keep the previous message sent over the edge, but rather keep the messages $\tilde{\delta}_{i \to j}$ sent in both directions; this more refined bookkeeping is necessary for dividing by the correct message following the approximation. This algorithm is called *expectation propagation* (EP) for reasons that are explained later.

expectation propagation

Example 11.8

To understand the behavior of this algorithm, consider the application of belief-update message propagation to example 11.7. Suppose we initialize all messages to 1 *and use updates of the form equation (11.39). We start by propagating a message* $\tilde{\sigma}_{1 \to 2} = \text{M-project-distr}_{1,2}(\tilde{\beta}_1)$ *from* $\boldsymbol{C}_1$ *to* $\boldsymbol{C}_2$. *Because* $\tilde{\delta}_{1 \to 2}$ *at this stage is 1, the resulting update is exactly the one we discussed in example 11.7. If we now perform propagation from* $\boldsymbol{C}_2$ *to* $\boldsymbol{C}_1$, *we get the message* $\tilde{\delta}_{2 \to 1}$ *derived in example 11.7, multiplied by a constant (since* $\tilde{\delta}_{1 \to 2}$ *is uniform). At this point, as we discussed, the clique beliefs* β_1 *are a fairly reasonable approximation to the posterior.*

Using the revised update rule, we now project $\tilde{\beta}_1$, *and then divide by* $\tilde{\delta}_{2 \to 1}$:

$$\tilde{\delta}_{1 \to 2} \leftarrow \frac{\text{M-project-distr}_{1,2}(\tilde{\beta}_1)}{\tilde{\delta}_{2 \to 1}}.$$

This quotient, which is thensubsequently used to update $\boldsymbol{C}_2$, *is very different from the previous update* $\tilde{\delta}_{2 \to 1}$. *Specifically, The marginal* $\tilde{\beta}_1$ *at this stage puts a posterior of* $0.642 + 0.031 = 0.673$ *on* a^1, *and* 0.642 *on* b^1. *To avoid double-counting the contribution of* $\tilde{\delta}_{2 \to 1}$, *we need to divide this marginal by this message. After we normalize messages, we obtain:*

$$\begin{aligned}
\tilde{\delta}_{1 \to 2}[a^1] &\leftarrow \frac{\frac{0.673}{0.904}}{\frac{0.673}{0.904} + \frac{0.327}{0.086}} = \frac{0.744}{4.15} = 0.179 \\
\tilde{\delta}_{1 \to 2}[a^0] &\leftarrow \frac{\frac{0.327}{0.086}}{\frac{0.673}{0.904} + \frac{0.327}{0.086}} = \frac{3.406}{4.15} = 0.821 \\
\tilde{\delta}_{1 \to 2}[b^1] &\leftarrow \frac{\frac{0.642}{0.173}}{\frac{0.642}{0.173} + \frac{0.358}{0.827}} = \frac{3.710}{4.413} = 0.895 \\
\tilde{\delta}_{1 \to 2}[b^0] &\leftarrow \frac{\frac{0.358}{0.827}}{\frac{0.642}{0.173} + \frac{0.358}{0.827}} = \frac{0.433}{4.144} = 0.105.
\end{aligned}$$

Recall that this message can be viewed as a "correction term" for the sepset marginals, relative to the message $\tilde{\delta}_{2\to 1}$. Thus, its effect is to reduce the support for a^1 (which was very high in $\tilde{\beta}_2$), and at the same time to increase the support for b^1.

Propagating this message to $\boldsymbol{C}_2$ and updating the clique beliefs $\tilde{\beta}_2$, we get a normalized factor of:

$$\begin{aligned}
\tilde{\beta}_2(c^0, d^0) &\leftarrow 0.031 \\
\tilde{\beta}_2(c^0, d^1) &\leftarrow 0.397 \\
\tilde{\beta}_2(c^1, d^0) &\leftarrow 0.317 \\
\tilde{\beta}_2(c^1, d^1) &\leftarrow 0.254.
\end{aligned}$$

These beliefs are closer to the exact marginals in that they distribute some of the mass that was previously assigned to c^1, d^0 to two other cases. However, they are still quite far from the exact marginal. ■

This example demonstrates several issues. First, there is significant difference between the two update rules. After incorporating $\tilde{\delta}_{2\to 1}$, the message from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ is readjusted to account for the new information, leading to a different approximation and hence a different update $\tilde{\delta}_{1\to 2}$. Second, unlike sum-product propagation, belief update propagation does not generally converge within two rounds, even in a clique tree. In fact, an immediate question is whether these iterations converge at all. And if they do, is there anything we can say about the convergence points? As we show in section 11.4.5, the answers to these questions are very similar to the answers we got in the case of cluster-graph belief propagation.

11.4.4 Expectation Propagation

So far, our discussion of approximate message passing has focused on a particular type of approximation: approximating a complex joint distribution as a product of small factors. However, the same ideas are applicable to a broad range of approximations. We now consider this process from a more general perspective, which will allow us to use a wider range of structured approximation in messages. Moreover, as we will see, this generalized form simplifies the variational analysis of this approach.

exponential family

The framework we use is based on the idea of the *exponential family*, as presented in chapter 8. As we discussed there, the exponential families are a general class of distributions, that contains many of the distributions of interest. Recall that a family of distributions $\mathcal{Q}$ is in the exponential family if it can be defined by two functions: the sufficient statistic function $\tau(\boldsymbol{x})$, and the natural parameter function $\mathtt{t}(\boldsymbol{\theta})$, so that any distribution Q in the family can be written as

$$Q(\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left\{\langle \tau(\boldsymbol{x}), \mathtt{t}(\boldsymbol{\theta})\rangle\right\},$$

where $\boldsymbol{\theta}$ is a set of parameters that specify the particular member of the family.

To simplify the discussion we will focus on linear exponential families, where $\mathtt{t}(\boldsymbol{\theta}) = \boldsymbol{\theta}$. Recall that linear exponential families include Markov networks (and consequently chordal Bayesian networks).

exponential family messages

As we now show, the approximate message passing approach described earlier applies when

Algorithm 11.6 The message passing step in the expectation propagation algorithm. The algorithm performs approximate message propagation by projecting expected sufficient statistics.

Procedure M-Project-Distr (
  $\mathcal{Q}$, // target exponential family for projection
  $\vec{\phi}$ // Factor set
)
1 $\boldsymbol{X} \leftarrow$ *Scope*$[\vec{\phi}]$ // Variables in factor set
2 $\bar{\tau} \leftarrow \boldsymbol{E}_{\boldsymbol{x} \sim \prod_{\phi \in \vec{\phi}}} \left[\tau_{\mathcal{Q}_{i,j}}(\boldsymbol{x}) \right]$
3 // Compute expectation of sufficient statistics relative to distribution defined by product of factors
4 $\theta \leftarrow$ M-project$(\bar{\tau})$
5 **return** (θ)

we choose to approximate messages by distributions from (linear) exponential families. Specifically, assume that we restrict each sepset $\boldsymbol{S}_{i,j}$ to be represented within an exponential family $\mathcal{Q}_{i,j}$ defined by a sufficient statistics function $\tau_{i,j}$. When performing message passing from $\boldsymbol{C}_i$ to $\boldsymbol{C}_j$, we compute the marginal of $\tilde{\beta}_i$, usually represented as a factor set $\vec{\phi}_i$, and project it into $\mathcal{Q}_{i,j}$ using the *M-projection* operator M-project-distr$_{i,j}$. This computation is often done using inference procedure that takes into account the structure of $\tilde{\beta}_i$ as a factor set.

M-projection

It turns out that the entire message passing operation can be formulated cleanly within this framework. If we are using an exponential family to represent our messages, then both the approximate clique marginal $\tilde{\sigma}_{i \to j}$ and the previous message $\tilde{\delta}_{j \to i}$ can be represented in the exponential form. Thus, if we ignore normalization factors, we have:

$$\begin{aligned}
\tilde{\sigma}_{i \to j} &\propto \exp\left\{ \langle \theta_{\tilde{\sigma}_{i \to j}}, \tau_{i,j}(\boldsymbol{s}_{i,j}) \rangle \right\} \\
\tilde{\delta}_{j \to i} &\propto \exp\left\{ \langle \theta_{\tilde{\delta}_{j \to i}}, \tau_{i,j}(\boldsymbol{s}_{i,j}) \rangle \right\} \\
\tilde{\delta}_{i \to j} &= \frac{\tilde{\sigma}_{i \to j}}{\tilde{\delta}_{j \to i}} \propto \exp\left\{ \langle (\theta_{\tilde{\sigma}_{i \to j}} - \theta_{\tilde{\delta}_{j \to i}}), \tau_{i,j}(\boldsymbol{s}_{i,j}) \rangle \right\},
\end{aligned}$$

where $\theta_{\tilde{\sigma}_{i \to j}}$ and $\theta_{\tilde{\delta}_{i \to j}}$ are the parameters of the messages $\tilde{\sigma}_{i \to j}$ and $\tilde{\delta}_{i \to j}$ respectively.

Since these messages are in an exponential family, it suffices to represent each of them by the parameters that describe them. We can then view propagation steps as updating these parameters. Specifically, we can rewrite the update step in line 4 of EP-Message as

$$\theta_{\tilde{\delta}_{i \to j}} \leftarrow (\theta_{\tilde{\sigma}_{i \to j}} - \theta_{\tilde{\delta}_{j \to i}}). \tag{11.40}$$

Note that, in the case of exact inference in discrete networks, the original update and the one using the exponential family representation are essentially identical, since the exponential family representation of factors is of the same size as the factor. Indeed, the standard update is often performed in a logarithmic representation (for reasons of numerical stability; see box 10.A), which gives rise precisely to the exponential family update.

The final issue we must address is the construction of the exponential-family representation of $\tilde{\sigma}_{i \to j}$ in line 1 of the algorithm. Recall that this process involves the M-projection of $\tilde{\beta}_i$

onto $\mathcal{Q}_{i,j}$. As we discussed in section 8.5, the M-projection of a distribution P within an exponential family $\mathcal{Q}$ is the distribution $Q \in \mathcal{Q}$, which defines the same expectation over the sufficient statistics as defined by P. In that section, we described a two-phase procedure for computing this approximating distribution: We compute the expected sufficient statistics induced by P, and then find the parameters for a distribution $Q \in \mathcal{Q}$ that induces the same expected sufficient statistics. We can apply this approach to define a general procedure for performing the operation M-project-distr$_{i,j}(\vec{\phi}_i)$ for a general exponential family. We first compute the expected expectation of $\tau_{i,j}$ according to $\tilde{\beta}_i$. We then find the distribution within $\mathcal{Q}_{i,j}$ that gives rise to the same expected sufficient statistics. This step is accomplished by the application of the function M-project that takes a vector of sufficient statistics and returns a parameter vector in the exponential family that induces precisely the expected sufficient statistics $\bar{\tau}_{i,j}$. This function is shown in algorithm 11.6. The use of the expectation step in computing the messages is the basis for the name *expectation propagation*, which describes the general procedure for any member of the exponential family.

expectation propagation

As we have already discussed, dividing by the previous message corresponds to subtraction of the parameters. Thus, overall, we obtain the following *EP message passing* step:

EP message passing

$$\theta_{\tilde{\delta}_{i\to j}} \leftarrow \text{M-project}_{i,j}(\mathbf{E}_{\boldsymbol{S}_{i,j}\sim\tilde{\beta}_i}[\tau_{i,j}(\boldsymbol{S}_{i,j})]) - \theta_{\tilde{\delta}_{j\to i}}. \tag{11.41}$$

How expensive are the two key steps in the expectation-propagation message passing procedure? The first step is computing the expectation $\mathbf{E}_{\boldsymbol{S}_{i,j}\sim\tilde{\beta}_i}[\tau_{i,j}]$. In the case of discrete networks, this step might be as expensive as the number of possible values of $\boldsymbol{C}_i$. However, as we saw in the previous section, in many cases we can use the structure of the factors that make up $\tilde{\beta}_i$ to perform this expectation much more efficiently. The second step is computing M-project on these factors. For some exponential families, this step is trivial, and can be done using a plug-in formula. In other families, this is a complex problem by itself. We will return to these issues in much greater detail in later chapters (particularly chapter 19 and chapter 20). Usually, when we design an approximation algorithm, we choose an exponential family for which this second step is easy.

The factored distributions we discussed earlier are perhaps the simplest example of an exponential family that we can use in this algorithm. However, other representations also fall into this class.

Example 11.9

Consider using a chain network to approximate each message, as in figure 11.15b. In example 8.16 and exercise 8.6 we showed that this class of distributions is a linear family and constructed the function M-project *for it. Following the derivation, suppose the variables in the sepset are $X_1, \ldots, X_k$ and we want to represent messages using the network structure $X_1 \to X_2 \to \ldots \to X_k$. In example 8.16, we showed that the expected sufficient statistics are summarized in the vector of indicators comprising: $\mathbf{1}\{x_i^j\}$ for $i = 1, \ldots, k$ and $x_i^j \in Val(X_i)$; and $\mathbf{1}\{x_i^j, x_{i+1}^\ell\}$ for $i = 1, \ldots, k-1$, $x_i^j \in Val(X_i)$, $x_i^\ell \in Val(X_{i+1})$. Once we have the expected value of these statistics, we can reconstruct the distribution $Q(X_{i+1} \mid X_i)$ as described in exercise 8.6. Given these subprocedures, the remaining propagation steps have been described earlier.*

If we consider a chain approximation to grid network, we can use the chains as in figure 11.13b. In this case, the main cost of a propagation step is the projection. The messages incoming to a cluster consists of univariate beliefs and pairwise beliefs along the column. When combined with

the clique factors, these result in a ladder-like network (shown in figure 11.13b). As we discussed, we can a build a (nested) clique tree for this network that involves clusters of at most three variables. Thus, we can perform inference efficiently on this nested clique tree to compute the expectations on pairwise beliefs in the outgoing column, and then use those expectations to reconstruct parameters. ■

The view of the expectation propagation algorithm in these terms allows us to understand the scope of this approach. We can apply this approach to any class of distributions in the linear exponential family for which the computation of expected sufficient statistics and the M-projection operation can be implemented effectively. Note that not every class of distributions we have discussed obeys these two restrictions. In particular, Markov networks are in the linear exponential family, but they do not have an effective M-projection procedure. Thus, a general Markov network structure is not often used to represent messages in the expectation propagation algorithm. Conversely, Bayesian networks are not in the linear exponential family (see section 8.3.2). One might argue that Bayesian networks should be usable, since the M-projection operation can be implemented analytically (see theorem 8.7), allowing the expectation propagation algorithm to be applied effectively.

This argument, however, brings up one important caveat regarding the expectation propagation update rule. In the rule, we subtract two sets of parameters: the parameters associated with the message from j to i are subtracted from the parameters obtained from M-projection operation for $\boldsymbol{C}_i$. For the classes of distributions that we considered earlier, the space of legal parameters Θ was the entire real space $\mathbb{R}^K$; this space is closed under subtraction, guaranteeing that the result of this update rule is a valid distribution in our space. This property does not hold for every exponential family; in particular, it is not the case for Bayesian networks with immoralities. Thus, we may end up in situations where the resulting parameters do not actually define a legal distribution in our space. We return to this issue when we discuss the application of expectation propagation to Gaussians in section 14.3.3, where it can give rise to severe problems. Thus, the most commonly used class of distributions in the expectation propagation algorithm is the class of low tree-width chordal graphs. Because these graphs are both Bayesian networks and Markov networks, they are both in the linear exponential family and admit an effective M-projection operation. The example of the chain distribution we used earlier falls into this category. In more general cases, these distributions are represented as clique trees, allowing them to be represented using a smaller set of factors: clique beliefs and sepset messages.

11.4.5 Variational Analysis

To define a variational principle for expectation propagation, we take an approach similar to the one we discussed in the context of cluster-graph belief propagation. Again, we consider an approximation $\boldsymbol{Q}$ that consists of a set of pseudo marginals. In fact, we use the same energy functional $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$.

The main difference from the case of cluster-graph belief propagation is that, in the current approximation, the cluster tree is *not* calibrated. As we project messages into an approximate form, we no longer ensure that beliefs of neighboring clusters agree on the joint distribution of the variables they share. Instead, we maintain a weaker property that depends on the nature of the approximation. For example, in the case where we use messages that are product of marginal distributions, intuition suggests that neighboring clusters will eventually agree on the marginal distributions of the variables in the sepset.

To gain better insight into the constraints we need, we start with reasoning about properties of convergence points of the algorithm. Suppose we iterate expectation-propagation belief update propagations until convergence. Now, consider two neighboring clusters i and j. Since the algorithm has converged, it follows that further updates do not change the cluster beliefs. Thus, the assignment of the expectation-propagation update rule of equation (11.41) becomes an equality for all clusters. We can then reason that

$$\text{M-project}_{i,j}(\boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\tilde{\beta}_i}[\tau_{i,j}]) = \theta_{\tilde{\delta}_{i\to j}} + \theta_{\tilde{\delta}_{j\to i}}.$$

A similar argument holds for the M-projection of the j cluster beliefs, $\text{M-project}_{i,j}(\boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\tilde{\beta}_j}[\tau_{i,j}])$. It follows that the projection of the two beliefs onto $\mathcal{Q}_{i,j}$ result in the same distribution.

This discussion suggests that we can pose the problem as optimizing the same objective as CTree-Optimize, except that we now replace the constraint equation (11.7) with an *expectation consistency constraint*:

expectation consistency constraint

$$\boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\mu_{i,j}}[\tau_{i,j}] = \boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\beta_j}[\tau_{i,j}]. \tag{11.42}$$

In general, $\tau_{i,j}$ is a vector, and so this equation defines a vector of constraints.

We now can define the optimization problem. It is identical to the optimization problems we already encountered, except that we relax the marginal consistency constraints.

EP-Optimize:

Find $\boldsymbol{Q}$
maximizing $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$
subject to

$$\boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\mu_{i,j}}[\tau_{i,j}] = \boldsymbol{E}_{\boldsymbol{S}_{i,j}\sim\beta_j}[\tau_{i,j}] \quad \forall (i\text{–}j) \in \mathcal{E}_\mathcal{T} \tag{11.43}$$

$$\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) = 1 \qquad \forall i \in \mathcal{V}_\mathcal{T} \tag{11.44}$$

$$\sum_{\boldsymbol{s}_{i,j}} \mu_{i,j}[\boldsymbol{s}_{i,j}] = 1 \quad \forall (i\text{–}j) \in \mathcal{E}_\mathcal{T} \tag{11.45}$$

$$\beta_i(\boldsymbol{c}_i) \geq 0 \qquad \forall i \in \mathcal{V}_\mathcal{T}, \boldsymbol{c}_i \in \mathit{Val}(\boldsymbol{C}_i). \tag{11.46}$$

Like CGraph-Optimize, this optimization problem is an approximation to the problem of optimizing the energy functional in two ways. First, the space over which we are optimizing is the set of pseudo-marginals $\boldsymbol{Q}$. Because our marginalization constraint only requires that two neighboring clusters agree on their expectations, they will generally not agree on the full marginals. Thus, in general, a solution to this problem will generally *not* correspond to the marginals of an actual distribution Q, even in the context of a clique tree. Second, because we can define the true energy function $F[\tilde{P}_\Phi, Q]$ only for coherent joint distributions, we must resort here to optimizing its factored form $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$. Although the two forms are equivalent for distributions over clique trees, they are not equivalent in this setting (since the exact energy functional is not even defined outside the space of coherent distributions Q). Thus, as in the case of the CGBP algorithms in the previous section, we are approximating both the objective and the optimization space.

The generalization of theorem 11.3 for this relaxed optimization problem follows using more or less the same proof structure, where we characterize the stationary points of the constrained objective using a set of *fixed-point equations* that define a message passing algorithm.

fixed-point equations

Theorem 11.8

Let $\boldsymbol{Q}$ be a set of beliefs such that $\mu_{i,j}$ is in the exponential family $\mathcal{Q}_{i,j}$ for all $(i,j) \in \mathcal{E}_\mathcal{T}$. Let M-project-distr$_{i,j}$ *be the M-projection operation into the family $\mathcal{Q}_{i,j}$. Then $\boldsymbol{Q}$ is a stationary point of* EP-Optimize *if and only if for every edge $(i,j) \in \mathcal{E}_\mathcal{T}$ there are auxiliary beliefs $\delta_{i\to j}(\boldsymbol{S}_{i,j})$ and $\delta_{j\to i}(\boldsymbol{S}_{i,j})$ so that*

$$\begin{aligned} \delta_{i\to j} &= \frac{\text{M-project-distr}_{i,j}(\beta_i)}{\delta_{j\to i}} \\ \beta_i &\propto \psi_i \cdot \prod_{j \in \text{Nb}_i} \delta_{j\to i} \\ \mu_{i,j} &\propto \delta_{j\to i} \cdot \delta_{i\to j}. \end{aligned} \tag{11.47}$$

Proof As in previous proofs of this kind, we define a Lagrangian that consists of the (approximate) energy functional $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ as well as Lagrange multiplier terms for each of the constraints. In our case, we have a vector of Lagrange multipliers for each of the constraints in equation (11.43).

$$\begin{aligned} \mathcal{J} = &\sum_{i \in \mathcal{V}_\mathcal{T}} \boldsymbol{E}_{\boldsymbol{C}_i \sim \beta_i}[\ln \psi_i] + \sum_{i \in \mathcal{V}_\mathcal{T}} \boldsymbol{H}_{\beta_i}(\boldsymbol{C}_i) - \sum_{(i,j) \in \mathcal{E}_\mathcal{T}} \boldsymbol{H}_{\mu_{i,j}}(\boldsymbol{S}_{i,j}) \\ &- \sum_i \sum_{j \in \text{Nb}_i} \vec{\lambda}_{j\to i} \cdot \left(\boldsymbol{E}_{\boldsymbol{S}_{i,j} \sim \mu_{i,j}}[\tau_{i,j}] - \boldsymbol{E}_{\boldsymbol{S}_{i,j} \sim \beta_j}[\tau_{i,j}] \right) \\ &- \sum_{i \in \mathcal{V}_\mathcal{T}} \lambda_i \left(\sum_{\boldsymbol{c}_i} \beta_i(\boldsymbol{c}_i) - 1 \right) - \sum_{(i,j) \in \mathcal{E}_\mathcal{T}} \lambda_{i,j} \left(\sum_{\boldsymbol{s}_{i,j}} \mu_{i,j}[\boldsymbol{s}_{i,j}] - 1 \right). \end{aligned}$$

Taking partial derivatives of J with respect to $\beta_i(\boldsymbol{c}_i)$ and $\mu_{i,j}[\boldsymbol{s}_{i,j}]$ and equating these derivatives to zero, we get the following equalities that must hold at a stationary point:

$$\begin{aligned} \beta_i(\boldsymbol{c}_i) &\propto \psi_i(\boldsymbol{c}_i) \prod_{j \in \text{Nb}_i} \exp\left\{ \vec{\lambda}_{j\to i} \cdot \tau_{i,j}(\boldsymbol{s}_{i,j}) \right\} \\ \mu_{i,j}[\boldsymbol{s}_{i,j}] &\propto \exp\left\{ (\vec{\lambda}_{j\to i} + \vec{\lambda}_{i\to j}) \cdot \tau_{i,j}(\boldsymbol{s}_{i,j}) \right\}. \end{aligned}$$

Note that $(\vec{\lambda}_{j\to i} + \vec{\lambda}_{i\to j})$ serves as the natural parameters of $\mu_{i,j}$ in its exponential form representation.

Moreover, the constraint of equation (11.43) implies that

$$\boldsymbol{E}_{\boldsymbol{S}_{i,j} \sim \mu_{i,j}}[\tau_{i,j}] = \boldsymbol{E}_{\boldsymbol{S}_{i,j} \sim \beta_j}[\tau_{i,j}].$$

Thus, using theorem 8.6, we conclude that $\mu_{i,j} = \text{M-project-distr}_{i,j}(\beta_i)$. By defining

$$\delta_{i\to j}[\boldsymbol{s}_{i,j}] \propto \exp\left\{ \vec{\lambda}_{i\to j} \cdot \tau_{i,}(\boldsymbol{s}_{i,j}) \right\},$$

we can verify that the statement of the theorem is satisfied. ∎

Theorem 11.8 shows that, if we perform EP belief update propagation until convergence, then we reach a stationary point of EP-Optimize. Thus, this result provides an optimization semantics for expectation-propagation message passing.

Our discussion of expectation propagation and the proof were posed in the context of linear exponential families. The same ideas can be extended to nonlinear families but require additional subtleties that we do not discuss.

11.4.6 Discussion

In this section, we presented an alternative approach for inference in large graphical models. Rather than modifying the global structure of the inference object, we modify the structure of the messages and how they are computed. Although we focused on the application of this approximation to clique trees, it is equally applicable in the context of cluster graphs. The message passing algorithms (both the sum-product algorithm and the belief update algorithm) can be used for passing messages between clusters in general graphs. Moreover, the variational analysis of section 11.4.5 also applies, essentially without change, to cluster graphs, using the same derivation as in section 11.3.

Note that the expectation propagation algorithm suffers from the same caveats we discussed in the previous section: **Iterations of EP message propagation are not guaranteed to induce monotonic improvements to the objective function, and the algorithm does not always converge. Moreover, even when the algorithm does converge, the clusters are only approximately calibrated: their marginals agree on the expectations of the sufficient statistics (say the individual marginals), but not on other aspects of the distribution (say marginals over pairs of variables). As a consequence, if we want to answer a query using the network, it may make a difference from which cluster we extract the answer.**

We presented expectation propagation in the context of its application to factored messages. In the simplest case, of fully factored messages, the messages are simply cluster marginals over individual variables. The similarity between this variant of expectation propagation and belief propagation is quite striking; indeed, one can simulate expectation propagation with fully factored messages using cluster-graph belief propagation with a particular factor graph structure (see exercise 11.23). The more general case of messages that are not fully factored (for example, figure 11.13b) is more complex, and they cannot be mapped directly to belief propagation in cluster graphs. However, a mapping does exist between expectation propagation in discrete networks with factorized messages and cluster-graph belief propagation with region graphs.

More important, however, is the fact that expectation propagation provides a general approach for dealing with distributions in the exponential family. It therefore provides message passing algorithms for a broad class of models. For example, we will see an application of expectation propagation to hybrid (continuous/discrete) graphical models in section 14.3.3. Its broad applicability makes expectation propagation an important component in the approximate inference toolbox.

11.5 Structured Variational Approximations

In the previous two sections, we examined approximations based on belief propagation. As we saw, both methods can be viewed as optimizing an approximate energy functional over the

class of pseudo-marginals. These pseudo-marginals generally do not correspond to a globally coherent joint distribution Q. **The *structured variational* approach aims to optimize the energy functional over a family $\mathcal{Q}$ of *coherent* distributions Q. This family is chosen to be computationally tractable, and hence it is generally not sufficiently expressive to capture all of the information in P_Φ.**

structured variational

More precisely, we aim to address the following maximization problem:

Structured-Variational:
Find $Q \in \mathcal{Q}$
maximizing $F[\tilde{P}_\Phi, Q]$

where $\mathcal{Q}$ is a given family of distributions. In these methods, we are using the exact energy functional $F[\tilde{P}_\Phi, Q]$, which satisfies theorem 11.2. Thus, maximizing the energy functional corresponds directly to obtaining a better approximation to P_Φ (in terms of $\mathbf{D}(Q\|P_\Phi)$).

The main parameter in this maximization problem is the choice of family $\mathcal{Q}$. This choice induces a trade-off. On one hand, families that are "simpler," that is, that can be described by a Bayesian network or a Markov network with small tree-width, allow more efficient inference. As we will see, simpler families also allow us to solve the maximization problem efficiently. On the other hand, if the family $\mathcal{Q}$ is too restrictive, then it cannot represent distributions that are good approximations of P_Φ, giving rise to a poor approximation Q. In either case, this family is generally chosen to have enough structure that allows inference to be tractable, giving rise to the name *structured variational* approximation.

structured variational

As we will see, the methods of this type differ from generalized belief propagation in several ways. They are guaranteed to lower-bound the log-partition function, and they also are guaranteed to converge.

11.5.1 The Mean Field Approximation

mean field

The first approach we consider is called the *mean field* approximation. As we will see, in many respects, it resembles the algorithm obtained using the Bethe approximation to the energy functional. In particular, the resulting algorithm performs message passing where the messages are distributions over single variables. As we will see, however, the form of the updates is somewhat different.

11.5.1.1 The Mean Field Energy

Unlike our presentation in earlier sections, we begin our discussion with the energy functional, and we derive the algorithm directly from analyzing it. The mean field algorithm finds the distribution Q, which is closest to P_Φ in terms of $\mathbf{D}(Q\|P_\Phi)$ within the class of distributions representable as a product of independent marginals:

$$Q(\mathcal{X}) = \prod_i Q(X_i). \tag{11.48}$$

On the one hand, the approximation of P_Φ as a fully factored distribution is likely to lose a lot of information in the distribution. On the other hand, this approximation is computationally

attractive, since we can easily evaluate any query on Q by a product over terms that involve the variables in the scope of the query. Moreover, to represent Q, we need only to describe the marginal probabilities of each of the variables.

energy functional

As in previous sections, the mean field algorithm is derived by considering fixed points of the *energy functional*. We thus begin by considering the form of the energy functional in equation (11.3) when Q has the form of a product distribution as in equation (11.48). We can then characterize its fixed points and thereby derive an iterative algorithm to find such fixed points.

The functional contains two terms. The first is a sum of terms of the form $\mathbf{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi]$, where we need to evaluate

$$\begin{aligned}\mathbf{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi] &= \sum_{\boldsymbol{u}_\phi} Q(\boldsymbol{u}_\phi) \ln \phi(\boldsymbol{u}_\phi) \\ &= \sum_{\boldsymbol{u}_\phi} \left(\prod_{X_i \in \boldsymbol{U}_\phi} Q(x_i) \right) \ln \phi(\boldsymbol{u}_\phi).\end{aligned}$$

As shown, we can use the form of Q to compute $Q(\boldsymbol{u}_\phi)$ as a product of marginals, allowing the evaluation of this term to be performed in time linear in the number of values of $\boldsymbol{U}_\phi$. Because this cost is linear in the description size of the factors of P_Φ, we cannot expect to do much better.

As we saw in section 8.4.1, the term $\mathbf{H}_Q(\mathcal{X})$ also decomposes in this case.

Corollary 11.3

If $Q(\mathcal{X}) = \prod_i Q(X_i)$, then

$$\mathbf{H}_Q(\mathcal{X}) = \sum_i \mathbf{H}_Q(X_i). \tag{11.49}$$

Thus, the energy functional for a fully factored distribution Q can be rewritten simply as a sum of expectations, each one over a small set of variables. Importantly, the complexity of this expression depends on the size of the factors in P_Φ, and *not* on the topology of the network. Thus, the energy functional in this case can be represented and manipulated effectively, even in networks that would require exponential time for exact inference.

Example 11.10

Continuing our running example, consider the form of the mean field energy for a 4×4 grid network. Based on our discussion, we see that it has the form

$$\begin{aligned}F[\tilde{P}_\Phi, Q] = &\sum_{i \in \{1,2,3\}, j \in \{1,2,3,4\}} \mathbf{E}_Q[\ln \phi(A_{i,j}, A_{i+1,j})] \\ &+ \sum_{i \in \{1,2,3,4\}, j \in \{1,2,3\}} \mathbf{E}_Q[\ln \phi(A_{i,j}, A_{i,j+1})] \\ &+ \sum_{i \in \{1,2,3,4\}, j \in \{1,2,3,4\}} \mathbf{H}_Q(A_{i,j}).\end{aligned}$$

We see that the energy functional involves only expectations over single variables and pairs of neighboring variables. The expression has the same general form for an $n \times n$ grid. Thus, although the tree-width of an $n \times n$ grid is exponential in n, the energy functional can be represented and computed in cost $O(n^2)$; that is, in a time linear in the number of variables. ■

11.5.1.2 Maximizing the Energy Functional: Fixed-point Characterization

The next step is to consider the task of optimizing the energy function: finding the distribution Q for which this energy functional is maximized:

Mean-Field:
Find $\{Q(X_i)\}$
maximizing $F[\tilde{P}_\Phi, Q]$
subject to

$$Q(\mathcal{X}) = \prod_i Q(X_i) \tag{11.50}$$

$$\sum_{x_i} Q(x_i) = 1 \quad \forall i. \tag{11.51}$$

To simplify notation, from now on we use $\boldsymbol{X}_{-i}$ to denote $\mathcal{X} - \{X_i\}$.

Note that, unlike the cluster-graph belief propagation algorithms of section 11.3 and the expectation propagation algorithm of section 11.4, here we are not approximating the objective. We are approximating only the optimization space by selecting a space of distributions $\mathcal{Q}$ that generally does not contain our original distribution P_Φ.

As with the previous optimization problems in this chapter, we use the method of Lagrange multipliers to derive a characterization of the stationary points of $F[\tilde{P}_\Phi, Q]$. However, the structure of Q allows us to consider the optimal value of each component (that is, marginal distribution) given the rest. (This iterative optimization procedure was not feasible in cluster trees and graphs due to constraints that relate different beliefs.)

fixed-point equations

We now provide a set of *fixed-point equations* that characterize the stationary points of the mean field optimization problem:

Theorem 11.9 *The distribution $Q(X_i)$ is a local maximum of* Mean-Field *given $\{Q(X_j)\}_{j \neq i}$ if and only if*

$$Q(x_i) = \frac{1}{Z_i} \exp\left\{ \sum_{\phi \in \Phi} \boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i] \right\}, \tag{11.52}$$

conditional expectation

where Z_i is a local normalizing constant and $\boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i]$ is the conditional expectation *given the value x_i*

$$\boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i] = \sum_{\boldsymbol{u}_\phi} Q(\boldsymbol{u}_\phi \mid x_i) \ln \phi(\boldsymbol{u}_\phi).$$

Proof The proof of this theorem relies on proving the fixed-point characterization of the individual marginal $Q(X_i)$ in terms of the other components, $Q(X_1), \ldots, Q(X_{i-1}), Q(X_{i+1}), \ldots, Q(X_n)$, as specified in equation (11.52).

We first consider the restriction of our objective $F[\tilde{P}_\Phi, Q]$ to those terms that involve $Q(X_i)$:

$$F_i[Q] = \sum_{\phi \in \Phi} \boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi] + \boldsymbol{H}_Q(X_i). \tag{11.53}$$

To optimize $Q(X_i)$, we define the Lagrangian that consists of all terms in $F[\tilde{P}_\Phi, Q]$ that involve $Q(X_i)$

$$L_i[Q] = \sum_{\phi \in \Phi} \boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi] + \boldsymbol{H}_Q(X_i) + \lambda \left(\sum_{x_i} Q(x_i) - 1 \right).$$

The Lagrange multiplier λ corresponds to the constraint that $Q(X_i)$ is a distribution. We now take derivatives with respect to $Q(x_i)$. The following result plays an important role in the remainder of the derivation:

Lemma 11.1 *If $Q(\mathcal{X}) = \prod_i Q(X_i)$ then, for any function f with scope $\boldsymbol{U}$,*

$$\frac{\partial}{\partial Q(x_i)} \boldsymbol{E}_{\boldsymbol{U} \sim Q}[f(\boldsymbol{U})] = \boldsymbol{E}_{\boldsymbol{U} \sim Q}[f(\boldsymbol{U}) \mid x_i].$$

The proof of this lemma is left as an exercise (see exercise 11.24).

Using this lemma, and standard derivatives of entropies, we see that

$$\frac{\partial}{\partial Q(x_i)} L_i = \sum_{\phi \in \Phi} \boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i] - \ln Q(x_i) - 1 + \lambda.$$

Setting the derivative to 0, and rearranging terms, we get that

$$\ln Q(x_i) = \lambda - 1 + \sum_{\phi \in \Phi} \boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i].$$

We take exponents of both sides and renormalize; because λ is constant relative to x_i, it drops out in the renormalization, so that we obtain the formula in equation (11.52).

This derivation, by itself, shows only that the solution of equation (11.52) is a stationary point of equation (11.53). To prove that it is a maximum, we note that equation (11.53) is a sum of two terms: $\sum_{\phi \in \Phi} \boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi]$ is linear in $Q(X_i)$, given all the other components $Q(X_j)$; $\boldsymbol{H}_Q(X_i)$ is a concave function in $Q(X_i)$. As a whole, given the other components of Q, the function F_i is concave in $Q(X_i)$, and therefore has a unique global optimum, which is easily verified to be equation (11.52) rather than any of the extremal points. ■

From this it follows that:

Corollary 11.4 *The distribution Q is a stationary point of* Mean-Field *if and only if, for each X_i, equation (11.52) holds.*

In contrast to theorem 11.9, this result only provides a characterization of stationary points of the objective, and not necessarily of its optima. The stationary points include local maxima, local minima, and saddle points. The reason for the difference is that, although each "coordinate" $Q(X_i)$ is guaranteed to be locally maximal given the others, the direction that locally improves the objective may require a coordinated change in several components. We return to this point in section 11.5.1.3.

We now move to interpreting this characterization. The key term in equation (11.52) is the argument in the expectation. We can prove the following property.

Corollary 11.5

In the mean field approximation, $Q(X_i)$ is locally optimal only if

$$Q(x_i) = \frac{1}{Z_i} \exp\left\{\boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}[\ln P_\Phi(x_i \mid \boldsymbol{X}_{-i})]\right\} \tag{11.54}$$

where Z_i is a normalizing constant.

Proof Recall that $\tilde{P}_\Phi = \prod_{\phi \in \Phi} \phi$ is the unnormalized measure defined by Φ. Due to the linearity of expectation:

$$\sum_{\phi \in \Phi} \boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i] = \boldsymbol{E}_{\mathcal{X} \sim Q}\left[\ln \tilde{P}_\Phi(X_i, \boldsymbol{X}_{-i}) \mid x_i\right].$$

Because Q is a product of marginals, we can rewrite $Q(\boldsymbol{X}_{-i} \mid x_i) = Q(\boldsymbol{X}_{-i})$, and get that:

$$\boldsymbol{E}_{\mathcal{X} \sim Q}\left[\ln \tilde{P}_\Phi(X_i, \boldsymbol{X}_{-i}) \mid x_i\right] = \boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}\left[\ln \tilde{P}_\Phi(x_i, \boldsymbol{X}_{-i})\right].$$

Using properties of conditional distributions, it follows that:

$$\tilde{P}_\Phi(x_i, \boldsymbol{X}_{-i}) = Z P_\Phi(x_i, \boldsymbol{X}_{-i}) = Z P_\Phi(\boldsymbol{X}_{-i}) P_\Phi(x_i \mid \boldsymbol{X}_{-i}).$$

We conclude that

$$\sum_{\phi \in \Phi} \boldsymbol{E}_{\mathcal{X} \sim Q}[\ln \phi \mid x_i] = \boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}[\ln P_\Phi(x_i \mid \boldsymbol{X}_{-i})] + \boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}[\ln P_\Phi(\boldsymbol{X}_{-i}) Z].$$

Plugging this equality into the update equation equation (11.52), we get that

$$Q(x_i) = \frac{1}{Z_i} \exp\left\{\boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}[\ln P_\Phi(x_i \mid \boldsymbol{X}_{-i})]\right\} \exp\left\{\boldsymbol{E}_{\boldsymbol{X}_{-i} \sim Q}[\ln P_\Phi(\boldsymbol{X}_{-i}) Z]\right\}.$$

The term $\ln P_\Phi(\boldsymbol{X}_{-i}) Z$ does not depend on the value of x_i. Recall that when we multiply a belief by a constant factor, it does not change the distribution Q; in fact, as we renormalize the distribution at the end to sum to 1, this constant will be "absorbed" into the normalizing function, to achieve normalization. Thus, we can simply ignore this term, thereby achieving the desired conclusion. We note that this type of algebraic manipulation will prove useful multiple times throughout this section. ■

This corollary shows that $Q(x_i)$ is the *geometric average* of the conditional probability of x_i given all other variables in the domain. The average is based on the probability that Q assigns to all possible assignments to the variables in the domain. In this sense, the mean field approximation requires that the marginal of X_i be "consistent" with the marginals of other variables.

Note that, in P_Φ, we can also represent the marginal of X_i as an average:

$$P_\Phi(x_i) = \sum_{\boldsymbol{x}_{-i}} P_\Phi(\boldsymbol{x}_{-i}) P_\Phi(x_i \mid \boldsymbol{x}_{-i}) = \boldsymbol{E}_{\boldsymbol{X}_{-i} \sim P_\Phi}[P_\Phi(x_i \mid \boldsymbol{X}_{-i})]. \tag{11.55}$$

This average is an arithmetic average, whereas the one used in the mean field approximation is a geometric average. In general, the latter tends to lead to marginals that are more sharply peaked than the original marginals in P_Φ. More significant, however, is the fact that the expectations in equation (11.55) are taken relative to P_Φ, whereas the ones in equation (11.54) are taken relative to the approximation Q. Thus, this similarity does not imply as a consequence that our approximation in Q to the marginals in P_Φ is a good one.

11.5.1.3 Maximizing the Energy Functional: The Mean Field Algorithm

How do we convert the fixed-point equation of equation (11.52) into an update algorithm? We start by observing that if $X_i \notin \mathit{Scope}[\phi]$ then $\mathbf{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi \mid x_i] = \mathbf{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi]$. Thus, expectation terms on such factors are independent of the value of X_i. Consequently, we can absorb them into the normalization constant Z_i and get the following simplification.

Corollary 11.6

In the mean field approximation, $Q(X_i)$ is locally optimal only if

$$Q(x_i) = \frac{1}{Z_i} \exp\left\{ \sum_{\phi: X_i \in \mathit{Scope}[\phi]} \mathbf{E}_{(\boldsymbol{U}_\phi - \{X_i\}) \sim Q}[\ln \phi(\boldsymbol{U}_\phi, x_i)] \right\}. \tag{11.56}$$

where Z_i is a normalizing constant.

This representation shows that $Q(X_i)$ has to be consistent with the expectation of the potentials in which it appears. In our grid network example, this characterization implies that $Q(A_{i,j})$ is a product of four terms measuring its interaction with each of its four neighbors:

$$Q(a_{i,j}) = \frac{1}{Z_{i,j}} \exp\left\{ \begin{array}{l} \sum_{a_{i-1,j}} Q(a_{i-1,j}) \ln(\phi(a_{i-1,j}, a_{i,j})) + \\ \sum_{a_{i,j-1}} Q(a_{i,j-1}) \ln(\phi(a_{i,j-1}, a_{i,j})) + \\ \sum_{a_{i+1,j}} Q(a_{i+1,j}) \ln(\phi(a_{i,j}, a_{i+1,j})) + \\ \sum_{a_{i,j+1}} Q(a_{i,j+1}) \ln(\phi(a_{i,j}, a_{i,j+1})) \end{array} \right\}. \tag{11.57}$$

Each term is a (geometric) average of one of the potentials involving $A_{i,j}$. For example, the final term in the exponent represents a geometric average of the potential between $A_{i,j}$ and $A_{i,j+1}$, averaged using the distribution $Q(A_{i,j+1})$.

The characterization of corollary 11.6 provides tools for developing an algorithm to maximize $F[\tilde{P}_\Phi, Q]$. For example, examining equation (11.57), we see that we can easily evaluate the term within the exponential by considering each of $A_{i,j}$'s neighbors and computing the interaction between the values that neighbor can take and possible values of $A_{i,j}$. Moreover, in this example, we see that $Q(A_{i,j})$ does *not* appear on the right-hand side of the update rule. Thus, we can choose $Q(A_{i,j})$, which satisfies the required equality by assigning it to the term denoted by the right-hand side of the equation.

This last observation is true in general. All the terms on the right-hand side of equation (11.56) involve expectations of variables other than X_i, and do not depend on the choice of $Q(X_i)$. We can achieve equality simply by evaluating the exponential terms for each value x_i, normalizing the results to sum to 1, and then assigning them to $Q(X_i)$. As a consequence, we reach the optimal value of $Q(X_i)$ in one easy step.

This last statement must be interpreted with some care. The resulting value for $Q(X_i)$ is its optimal value *given* the choice of all other marginals. Thus, this step optimizes our function relative only to a single coordinate in the space — the marginal of $Q(X_i)$. To optimize the function in its entirety, we need to optimize relative to all of the coordinates. We can embed this step in an iterated *coordinate ascent* algorithm, which repeatedly optimizes a single marginal at a time, given fixed choices to all of the others. The resulting algorithm is shown in algorithm 11.7. Importantly, a single optimization of $Q(X_i)$ does not usually suffice: a subsequent modification

Algorithm 11.7 The Mean-Field approximation algorithm

Procedure Mean-Field (
  Φ, // factors that define P_Φ
  Q_0 // Initial choice of Q
)
1 $Q \leftarrow Q_0$
2 *Unprocessed* $\leftarrow \mathcal{X}$
3 **while** *Unprocessed* $\neq \emptyset$
4 Choose X_i from *Unprocessed*
5 $Q_{old}(X_i) \leftarrow Q(X_i)$
6 **for** $x_i \in Val(X_i)$ **do**
7 $Q(x_i) \leftarrow \exp\left\{\sum_{\phi: X_i \in Scope[\phi]} \mathbf{E}_{(\boldsymbol{U}_\phi - \{X_i\}) \sim Q}[\ln \phi[\boldsymbol{U}_\phi, x_i]]\right\}$
8 Normalize $Q(X_i)$ to sum to one
9 **if** $Q_{old}(X_i) \neq Q(X_i)$ **then**
10 *Unprocessed* $\leftarrow$ *Unprocessed* $\cup \left(\cup_{\phi: X_i \in Scope[\phi]} Scope[\phi]\right)$
11 *Unprocessed* $\leftarrow$ *Unprocessed* $- \{X_i\}$
12 **return** Q

to another marginal $Q(X_j)$ may result in a different optimal parameterization for $Q(X_i)$. Thus, the algorithm repeats these steps until convergence. Note that, in practice, we do not test for equality in line 9, but rather for equality up to some fixed small-error tolerance.

A key property of the coordinate ascent procedure is that each step leads to an increase in the energy functional. **Thus, each iteration of** Mean-Field **results in a better approximation** Q **to the target density** P_Φ**, guaranteeing convergence.**

Theorem 11.10 *The* Mean-Field *iterations are guaranteed to converge. Moreover, the distribution* Q^* *returned by* Mean-Field *is a stationary point of* $F[\tilde{P}_\Phi, Q]$*, subject to the constraint that* $Q(\mathcal{X}) = \prod_i Q(X_i)$ *is a distribution.*

Proof We showed earlier that each iteration of Mean-Field is monotonically nondecreasing in $F[\tilde{P}_\Phi, Q]$. Because the energy functional is bounded, the sequence of distributions represented by successive iterations of Mean-Field must converge. At the convergence point the fixed-point equations of theorem 11.9 hold for all the variables in the domain. As a consequence, the convergence point is a stationary point of the energy functional. ■

As we discussed, the distribution Q^* returned by Mean-Field is not necessarily a local optimum of the algorithm. However, local minima and saddle points are not stable convergence points of the algorithm, in the sense that a small perturbation of Q followed by optimization will lead to a better convergence point. Because the algorithm is unlikely to accidentally land precisely on the unstable point and get stuck there, in practice, the convergence points of the algorithm are local maxima.

In general, however, the result of the mean field approximation is a local maximum, and not necessarily a global one.

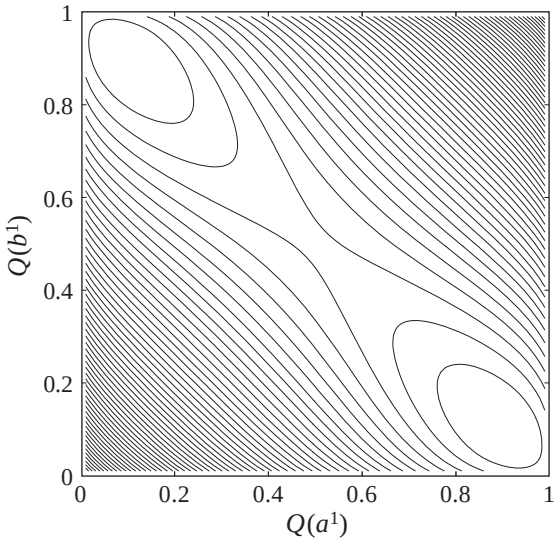


Figure 11.16 An example of a multimodal mean field energy functional landscape. In this network, $P(a, b) = 0.5 - \epsilon$ if $a \neq b$ and ϵ if $a = b$. The axes correspond to the mean field marginal for $Q(a^1)$ and $Q(b^1)$ and the contours show equi-values of the energy functional for different choices of these variational parameters. As we can see, there are two modes to the energy functional, one roughly corresponds to a^1, b^0 and the other one to a^0, b^1. In addition, there is a saddle point at $(0.5, 0.5)$.

Example 11.11

Consider a distribution P_Φ that is an approximate XOR (exclusive or) of two variables A and B, so that $P_\Phi(a, b) = 0.5 - \epsilon$ if $a \neq b$ and $P_\Phi(a, b) = \epsilon$ if $a = b$. Clearly, we cannot approximate P_Φ by a product of marginals, since such a product cannot capture the relationship between A and B. It turns out that if ϵ is sufficiently small, say 0.01, then the energy potential surface has two local maxima that correspond to the two cases where $a \neq b$. See figure 11.16. (For sufficiently large ϵ, such as 0.1, the mean field approximation has a single maximum point at the uniform distribution.) ■

We can use standard strategies, such as multiple random restarts, to try to avoid getting stuck in local maxima. However, these do not overcome the basic shortcoming of the mean field approximation, which is apparent in this example. The approximation cannot describe complex posteriors, such as the XOR posterior we discussed. And thus, we cannot expect it to give satisfactory approximations in these situations. To provide better approximations, we must use a richer class of distributions $\mathcal{Q}$, which has greater expressive power.

11.5.2 Structured Approximations

The mean field algorithm provides an easy approximation method. However, it is limited by forcing Q to be a very simple distribution. As we just saw, the fact that all variables are independent of each other in Q can lead to very poor approximations. Intuitively, if we

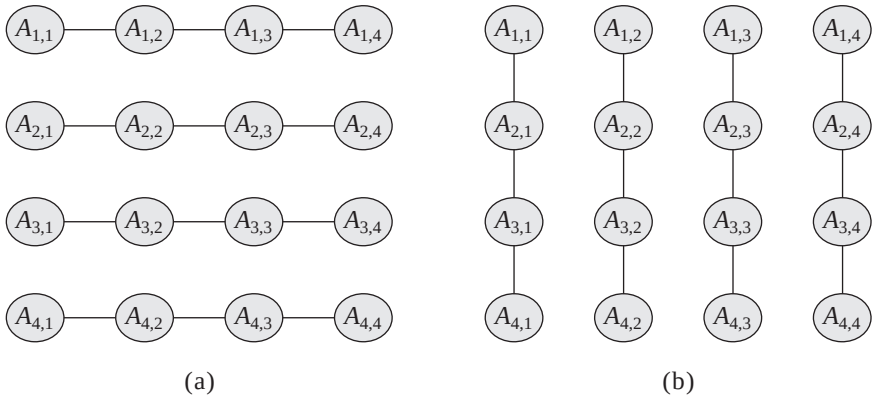


Figure 11.17 Two structures for variational approximation of a 4×4 grid network

use a distribution Q that can capture some of the dependencies in P_Φ, we can get a better approximation. Thus, we would like to explore the spectrum of approximations between the mean field approximation and exact inference.

A natural approach to get richer approximations that capture some of the dependencies in P_Φ is to use network structures of different complexity. By adding and removing edges from the network we can control the cost of inference in the approximating distribution and how well it captures dependencies in the target distribution. We can achieve this type of flexibility by using either Bayesian networks or Markov networks. Both types of networks lead to similar approximations, and so we focus on the undirected case, parameterized as Gibbs distributions (so that we are not restricted to factors over maximal cliques). Exercise 11.34 develops similar ideas using a Bayesian network approximation.

11.5.2.1 Fixed-Point Characterization

We now consider the form of the variational approximation when we are given a general form of Q as a Gibbs parametric family. Formally, we assume we are given a set of potential scopes $\{\boldsymbol{C}_j \subseteq \mathcal{X} : j = 1, \ldots, J\}$. We can then choose an approximation Q that has the form:

$$Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_{j=1}^{J} \psi_j, \tag{11.58}$$

where ψ_j is a factor with $\mathit{Scope}[\psi_j] = \boldsymbol{C}_j$.

Example 11.12

Consider again the grid network example. There are many possible approximating network structures we can choose that allow for efficient inference. As a concrete example, we might choose potential scopes $\{A_{1,1}, A_{1,2}\}$, $\{A_{1,2}, A_{1,3}\}, \ldots, \{A_{2,1}, A_{2,2}\}$, $\{A_{2,2}, A_{2,3}\}, \ldots$. That is, we preserve the dependencies between variables in the same row, but ignore the ones that relate different columns. Alternatively, we can consider an approximation that preserves dependencies along columns and ignores the dependencies between rows. As we can see in figure 11.17, in both cases,

the structure we use is a collection of independent chain structures. Exact inference with such structures is linear, and so the cost of inference is not much worse than in the mean field approximation. Clearly, we can also consider many other structures for the approximating distributions. These might introduce additional dependencies and can have higher cost in terms of inference. We will return to the question of what structure to use. ■

Assume that we decide on the form of the potentials for the approximating family $\mathcal{Q}$. As before, we consider the form of the energy functional for a distribution Q in this family. We then characterize the stationary points of the functional, and we use those to derive an iterative optimization algorithm.

As before, evaluating the terms that involve $\boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\ln \phi]$ requires performing expectations with respect to the variables in $Scope[\phi]$. Unlike the case of mean field approximation, the complexity of computing this expectation depends on the structure of the approximating distribution. However, we assume that we can solve this problem by exact inference (in the network corresponding to Q), using the methods we discussed in previous chapters.

As discussed in section 8.4.1, the entropy term in the energy functional also reduces to computing a similar set of expectation terms:

Proposition 11.5

If $Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_j \psi_j$, *then*

$$\boldsymbol{H}_Q(\mathcal{X}) = -\sum_{j=1}^{J} \boldsymbol{E}_{\boldsymbol{C}_j \sim Q}[\ln \psi_j(\boldsymbol{C}_j)] + \ln Z_Q.$$

Overall, we obtain the following form for the energy functional, for distributions Q in the family $\mathcal{Q}$:

$$F[\tilde{P}_\Phi, Q] = \sum_{k=1}^{K} \boldsymbol{E}_Q[\ln \phi_k] - \sum_{j=1}^{J} \boldsymbol{E}_Q[\ln \psi_j] + \ln Z_Q. \tag{11.59}$$

As before, the hard question is how to optimize the potential to get the best approximation. We solve this problem using the same general strategy we discussed in the context of the mean field approximation. First, we derive the fixed-point equations that hold when the approximation is a local maximum (or, more precisely, a stationary point) of the energy functional. We then use these fixed-point equations to help derive an optimization algorithm.

fixed-point equations

We derive the *fixed-point equations* by taking derivatives of $F[\tilde{P}_\Phi, Q]$ with respect to parameters of the distribution Q. In our case, the parameters will be an entry $\psi_j(\boldsymbol{c}_j)$ in each of the factors that define the distribution. We then set those equations to zero, obtaining the following result:

Theorem 11.11

If $Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_j \psi_j$, *then the potential* ψ_j *is a stationary point of the energy functional if and only if*

$$\psi_j(\boldsymbol{c}_j) \propto \exp\left\{ \boldsymbol{E}_Q\left[\ln \tilde{P}_\Phi \mid \boldsymbol{c}_j\right] - \sum_{k \neq j} \boldsymbol{E}_Q[\ln \psi_k \mid \boldsymbol{c}_j] - F[\tilde{P}_\Phi, Q] \right\}. \tag{11.60}$$

The proof is straightforward algebraic manipulation and is left as an exercise (exercise 11.26).

This theorem establishes a characterization of the fixed point as the difference between the expected value of logarithm of the original potentials and the expected value of the logarithm of the approximating potentials. The last term in equation (11.60) is the energy functional $F[\tilde{P}_\Phi, Q]$, which is independent of the assignment $\boldsymbol{c}_j$; thus, as we discussed in the proof of corollary 11.5, we can absorb this term into the normalization constant of the distribution and ignore it.

Corollary 11.7

If $Q(\mathcal{X}) = \frac{1}{Z_Q}\prod_j \psi_j$, then the potential ψ_j is a stationary point of the energy functional if and only if:

$$\psi_j(\boldsymbol{c}_j) \propto \exp\left\{ \boldsymbol{E}_Q\left[\ln \tilde{P}_\Phi \mid \boldsymbol{c}_j\right] - \sum_{k \neq j} \boldsymbol{E}_Q[\ln \psi_k \mid \boldsymbol{c}_j] \right\}. \tag{11.61}$$

As we show in section 11.5.2.3 and section 11.5.2.4, we can often exploit additional structure in Q to reduce further the complexity of the fixed-point equations, and hence of the resulting update steps. The following discussion, which describes the procedure of applying the fixed-point equations to find a stationary point of the energy functional, is orthogonal to these simplifications.

11.5.2.2 Optimization

Given a set of fixed-point equations as in equation (11.61), our task is to find a distribution Q that satisfies them. As in section 11.5.1, our strategy is based on the key observation that the factor ψ_j does not affect the right-hand side of the fixed-point equations defining its value: The first expectation, $\boldsymbol{E}_Q\left[\ln \tilde{P}_\Phi \mid \boldsymbol{c}_j\right]$, is conditioned on $\boldsymbol{c}_j$ and therefore does not depend on the parameterization of ψ_j. The same observation holds for the second expectation, $\boldsymbol{E}_Q[\ln \psi_k \mid \boldsymbol{c}_j]$, for any $k \neq j$. (Importantly, there is no such term for $k = j$ in the right-hand side.) Thus, we can use the same general approach as in Mean-Field: We can optimize each potential ψ_j, *given values for the other potentials*, by simply selecting ψ_j to satisfy the fixed-point equation. As for the case of mean field, this step is guaranteed to increase (or not decrease) the value of the objective; thus, the overall process is guaranteed to converge to a stationary point of the objective.

This last step requires that we perform inference in the approximating distribution Q to compute the requisite expectations. Although this step was also present (implicitly) in the mean field approximation, there the structure of the approximating distribution was trivial, and so the inference step involved only individual marginals. Here, we need to collect the expectation of several factors, and each of these requires that we compute expectations given different assignments to the factor of interest. (See exercise 11.27 for a discussion of how these expectations can be computed efficiently.) For a general distribution Q, even one with tractable structure, running inference in the corresponding network $\mathcal{H}_Q$ can be costly, and we may want to reduce the number of calls to the inference subroutine.

This observation leads to a question of how best to perform updates for several factors in Q. We can consider two strategies. The *sequential update* strategy is similar to our strategy in the mean field algorithm: We choose a factor ψ_j, apply the fixed-point equation to that

factor by running inference in $\mathcal{H}_Q$, update the distribution, and then repeat this process with another factor until convergence. The problematic aspect of this approach is that we need to perform inference after each update step. For example, if we are using cluster tree inference in the network $\mathcal{H}_Q$, the network parameterization changes after each update step, so we need to recalibrate the clique tree every time. Some of these steps can be made more efficient by selecting an appropriate order of updates and using dynamic programming (see exercise 11.27), but the process can still be quite expensive.

An alternative approach is the *parallel update* strategy, where we compute the right-hand side of our fixed-point equations (for example, equation (11.61)) simultaneously for each of the factors in Q. If we are using a cluster tree for inference, this process involves multiple queries from the same calibrated cluster tree. Thus, we can perform a single calibration step and use the resulting tree to reestimate all of our potentials. However, the different queries required all have different evidence; hence, it is not easy to obtain significant computational savings, and the algorithms needed are fairly tricky. Nevertheless, this approach might be less costly than recalibrating the clique tree J times.

On the other hand, the guarantees provided by these two update steps are different. For the sequential update strategy, we can prove that each update step is monotonic in the energy functional: each step maximizes the value of one potential given the values of all the others, and therefore is guaranteed not to decrease (and generally to increase) the energy functional. This monotonic improvement implies that iterations of sequential updates necessarily converge, generally to a local maximum. The issue of convergence is more complicated in the parallel update strategy. Because we update all the potentials at once, we have no guarantees that any fixed-point equation holds after the update; a value that was optimal for ψ_j with respect to the values of all other factors before the parallel update step is not necessarily optimal given their new values. As such, it is conceivable that parallel updates will not converge (for example, oscillate between two sets of values for the potentials). Such oscillations can generally be avoided using damped update steps, similar to these we discussed in the case of cluster-graph belief propagation (see box 11.B), but this modified procedure still does not guarantee convergence.

At this point, there is no generally accepted procedure for scheduling updates in variational methods, and different approaches are likely to be best for different applications.

11.5.2.3 Simplifying the Update Equations

Equation (11.61) provides a general characterization of the fixed points of the energy functional, for any approximating class of distributions $\mathcal{Q}$ obeying a particular factorization, as in equation (11.58). In many cases, we can exploit additional structure of the approximating class $\mathcal{Q}$ and of the distribution P_Φ to simplify significantly the form of these fixed-point equations and thereby make the update step more efficient.

The simplifications we describe take two forms. The first utilizes marginal independencies in $\mathcal{Q}$ to simplify the right-hand side of the fixed-point equation, equation (11.61), eliminating irrelevant terms. The second exploits interactions between the form of $\mathcal{Q}$ and the form of P_Φ to simplify the factorization of $\mathcal{Q}$, without loss in expressive power. Both simplifications allow the fixed-point updates to be performed more efficiently. We motivate each of the simplifications using an example, and then we present the general result.

Example 11.13

Once again, consider the 4×4 grid network. Assume that we approximate it by a "row" network that has the structure shown in figure 11.17a. This approximating network consists of four independent chains. Now we can apply the general form of the fixed-point equation equation (11.61) for a specific entry in our approximation, say:

$$\psi_{1,1}(a_{1,1},a_{1,2}) \propto \exp\left\{ \begin{array}{l} \boldsymbol{E}_Q\left[\ln \tilde{P}_\Phi \mid a_{1,1},a_{1,2}\right] \\ -\sum_{\substack{i=1,\ldots,4 \\ j=1,\ldots,3 \\ (i,j)\neq(1,1)}} \boldsymbol{E}_Q\left[\ln \psi_{(i,j)}(A_{i,j},A_{i,j+1}) \mid a_{1,1},a_{1,2}\right] \end{array} \right\}.$$

As in the proof of corollary 11.5, the expectation of $\ln \tilde{P}_\Phi$ is the sum of expectations of the logarithm of each of the potentials in Φ. Some of these terms, however, do not depend on the choice of value of $A_{1,1}, A_{1,2}$ we are evaluating. For example, because $A_{2,1}$ and $A_{2,2}$ are independent of $A_{1,1}, A_{1,2}$ in Q, we conclude that

$$\boldsymbol{E}_{\{A_{2,1},A_{2,2}\}\sim Q}[\ln \phi(A_{2,1},A_{2,2}) \mid a_{1,1},a_{1,2}] = \boldsymbol{E}_{\{A_{2,1},A_{2,2}\}\sim Q}[\ln \phi(A_{2,1},A_{2,2})].$$

Thus, this term will contribute the same value to each of the entries of $\psi(A_{1,1},A_{1,2})$, and can therefore be absorbed into the corresponding normalizing term. We can continue in this manner and remove all terms that are not dependent on the context of the factor we are interested in. Overall, we can remove any term $\boldsymbol{E}_Q[\ln \phi(A_{i,j},A_{i,j+1}) \mid a_{1,1},a_{1,2}]$ and any term $\boldsymbol{E}_Q[\ln \phi(A_{i,j},A_{i+1,j}) \mid a_{1,1},a_{1,2}]$ except those where $i=1$. Similarly, we can remove any term $\boldsymbol{E}_Q\left[\ln \psi_{(i,j)}(A_{i,j},A_{i,j+1}) \mid a_{1,1},a_{1,2}\right]$ except those where $i=1$. These simplifications result in the following update rule:

$$\psi_{1,1}(a_{1,1},a_{1,2}) \propto \exp\left\{ \begin{array}{l} \sum_{j=1,\ldots,3} \boldsymbol{E}_{\{A_{1,j},A_{1,j+1}\}\sim Q}\left[\ln \phi_{(1,j)}(A_{1,j},A_{1,j+1}) \mid a_{1,1},a_{1,2}\right] \\ +\sum_{j=1,\ldots,4} \boldsymbol{E}_{\{A_{1,j},A_{2,j}\}\sim Q}\left[\ln \phi_{(1,j)}(A_{1,j},A_{2,j}) \mid a_{1,1},a_{1,2}\right] \\ -\sum_{j=2,3} \boldsymbol{E}_{\{A_{1,j},A_{1,j+1}\}\sim Q}\left[\ln \psi_{(1,j)}(A_{1,j},A_{1,j+1}) \mid a_{1,1},a_{1,2}\right] \end{array} \right\}.$$

■

We can generalize this analysis to arbitrary sets of factors:

Theorem 11.12

If $Q(\mathcal{X}) = \frac{1}{Z_Q}\prod_j \psi_j$, then the potential ψ_j is locally optimal only if

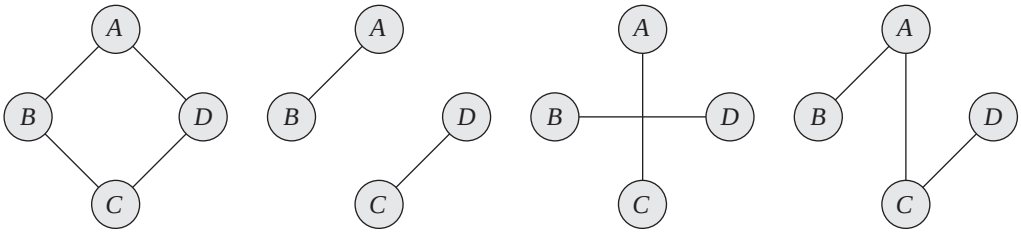
$$\psi_j(\boldsymbol{c}_j) \propto \exp\left\{ \sum_{\phi\in A_j} \boldsymbol{E}_{\mathcal{X}\sim Q}[\ln \phi \mid \boldsymbol{c}_j] - \sum_{\psi_k\in B_j} \boldsymbol{E}_{\mathcal{X}\sim Q}[\ln \psi_k \mid \boldsymbol{c}_j] \right\}, \tag{11.62}$$

where

$$A_j = \{\phi\in\Phi : Q \not\models (\boldsymbol{U}_\phi \perp \boldsymbol{C}_j)\}$$

and

$$B_j = \{\psi_k : Q \not\models (\boldsymbol{C}_k \perp \boldsymbol{C}_j)\} - \{\boldsymbol{C}_j\}.$$



(a) $\mathcal{H}_{P_\Phi}$ (b) $\mathcal{H}_{\mathcal{Q}_1}$ (c) $\mathcal{H}_{\mathcal{Q}_2}$ (d) $\mathcal{H}_{\mathcal{Q}_3}$

Figure 11.18 A diamond network and three possible approximating structures

.

Stated in words, this result shows that the parameterization of a factor $\psi_j(\boldsymbol{C}_j)$ depends only on factors in P_Φ and in Q whose scopes are not independent of $\boldsymbol{C}_j$ in $\mathcal{Q}$. This result, applied to example 11.13, provides us precisely with the simplification shown: only factors whose scopes intersect with the first row are relevant to $\psi_{1,1}(A_{1,1}, A_{1,2})$. Thus, we can use independence properties of the approximating family $\mathcal{Q}$ to simplify the right-hand side of equation (11.61) by removing irrelevant terms.

11.5.2.4 Simplifying the Family $\mathcal{Q}$

It turns out that a similar analysis allows us to simplify the form of the approximating family $\mathcal{Q}$ without loss in the quality of the approximation.

We start by considering a simple example.

Example 11.14

Consider again the four-variable pairwise Markov network of figure 11.18a, which is parameterized by the pairwise factors:

$$P_\Phi(A, B, C, D) \propto \phi_{AB}(A, B) \cdot \phi_{BC}(B, C) \cdot \phi_{CD}(C, D) \cdot \phi_{AD}(A, D).$$

Consider applying the variational approximation with the distribution

$$Q(A, B, C, D) = \frac{1}{Z_Q}\psi_1(A, B) \cdot \psi_2(C, D) \tag{11.63}$$

that has the structure shown in figure 11.18b. Using equation (11.62), we conclude that the fixed-point characterization of ψ_1 is

$$\psi_1(a, b) \propto \exp\{\boldsymbol{E}_Q[\ln \phi_{AB}(A, B) \mid a, b] + \boldsymbol{E}_Q[\ln \phi_{BC}(B, C) \mid a, b] + \boldsymbol{E}_Q[\ln \phi_{AD}(A, D) \mid a, b]\}.$$

Can we further simplify this equation? Consider the first term. Clearly, $\boldsymbol{E}_Q[\ln \phi_{AB}(A, B) \mid a, b] = \ln \phi_{AB}(a, b)$. What about the second term, $\boldsymbol{E}_Q[\ln \phi_{BC}(B, C) \mid a, b]$? To compute this expectation, we need to compute $Q(B, C \mid a, b)$. According to the structure of Q, we can see that

$$Q(B, C \mid a, b) = \begin{cases} Q(C) & \text{If } B = b \\ 0 & \text{otherwise.} \end{cases}$$

Thus, we conclude that

$$\boldsymbol{E}_{A,B,C\sim Q}[\ln \phi_{BC}(B,C) \mid a,b] = \boldsymbol{E}_{C\sim Q}[\ln \phi_{BC}(b,C)].$$

We can simplify the third term in exactly the same way, concluding that:

$$\psi_1(a,b) \propto \exp\left\{\ln \phi_{AB}(a,b) + \boldsymbol{E}_{C\sim Q}[\ln \phi_{BC}(b,C)] + \boldsymbol{E}_{D\sim Q}[\ln \phi_{AD}(a,D)]\right\}.$$

Setting $\psi_1'(a) = \exp\{\boldsymbol{E}_{D\sim Q}[\ln \phi_{AD}(a,D)]\}$ *and* $\psi_1''(b) = \exp\{\boldsymbol{E}_{C\sim Q}[\ln \phi_{BC}(b,C)]\}$*, we conclude that the optimal* ψ_1 *factorizes as a product of three factors:*

$$\psi_1(A,B) = \phi_{AB}(A,B) \cdot \psi_1'(A) \cdot \psi_1''(B).$$

Have we gained anything from this decomposition? First, we see that Q *preserves the original pairwise interaction term* $\phi(A,B)$ *from* P_Φ*. Moreover, the effect of the interactions between these variables and the rest of the network (*C *and* D *in this example) is summarized by a univariate factor for each of the variables. Thus,* Q *does not change the interaction between* A *and* B*.*

Applying the same set of arguments to ψ_2*, we conclude that we can rewrite* Q *as*

$$Q'(A,B,C,D) = \frac{1}{Z_Q}\phi_{AB}(A,B) \cdot \phi_{CD}(C,D) \cdot \psi_1'(A) \cdot \psi_1''(B) \cdot \psi_2'(C) \cdot \psi_2''(D). \quad (11.64)$$

The preceding discussion shows that the best approximation to P_Φ *within* $\mathcal{Q}$ *can be rewritten in the form of* Q'*. Thus, there is no point in using the more complicated form of the approximating family of equation (11.63); we may as well use the form of* Q' *in equation (11.64). Note that the form of* Q' *involves a product of a subset of the original factors, which we keep intact without change, and a set of new factors, which we need to optimize. In this example, instead of estimating two pairwise potentials, we estimate four univariate potentials, which utilize a smaller number of parameters.*

Moreover, the update equations for Q' *are simpler. Consider, for example, applying equation (11.62) for* ψ_1'*:*

$$\begin{aligned}\ln \psi_1'(a) \propto\ & \boldsymbol{E}_{B\sim Q'}[\ln \phi_{AB}(a,B) \mid a] + \boldsymbol{E}_{D\sim Q'}[\ln \phi_{AD}(a,D) \mid a] + \boldsymbol{E}_{B,C\sim Q'}[\ln \phi_{BC}(B,C) \mid a] \\ & - \boldsymbol{E}_{B\sim Q'}[\ln \phi_{AB}(a,B) \mid a] - \boldsymbol{E}_{B\sim Q'}[\ln \psi_1''(B) \mid a] \\ =\ & \boldsymbol{E}_{D\sim Q'}[\ln \phi_{AD}(a,D) \mid a] + \boldsymbol{E}_{B,C\sim Q'}[\ln \phi_{BC}(B,C) \mid a] - \boldsymbol{E}_{B\sim Q'}[\ln \psi_1''(B) \mid a].\end{aligned}$$

The terms involving $\boldsymbol{E}_{Q'}[\ln \phi_{AB} \mid a]$ *appear twice, once as a factor in* P_Φ *and once as a factor in* Q'*. These two terms cancel out, and we are left with the simpler update equation. Although this equation does not explicitly mention* ϕ_{AB}*, this factor participates in the computation of* $Q'(B \mid a)$ *that implicitly appears in* $\boldsymbol{E}_{B\sim Q'}[\ln \psi_1''(B) \mid a]$*.* ■

Note that this result is somewhat counterintuitive, since it shows that the interactions between A and B are captured by the original potential in P_Φ. Intuitively, we would expect the chain of influence A—D—C—B to introduce additional interactions between A and B that should be represented in Q. This is not the only counterintuitive result.

Example 11.15

Consider another approximating family for the same network, using the network structure shown in figure 11.18c. In this approximation, we have two pairwise factors, $\psi_1(A, C)$, and $\psi_2(B, D)$. Applying the same set of arguments as before, we can show that the update equation can be written as

$$\begin{aligned}\ln \psi_1(a, c) \quad \propto \quad & \boldsymbol{E}_{B \sim Q}[\ln \phi_{AB}(a, B)] + \boldsymbol{E}_{D \sim Q}[\ln \phi_{AD}(a, D)] \\ & + \boldsymbol{E}_{B \sim Q}[\ln \phi_{BC}(B, c)] + \boldsymbol{E}_{D \sim Q}[\ln \phi_{CD}(c, D)].\end{aligned}$$

Thus, we can factorize ψ_1 into two factors, one with a scope of A and the other with C

$$\psi_1(A, C) = \psi_1'(A) \cdot \psi_1''(C).$$

In other words, the approximation in this case is equivalent to the mean field approximation. This result shows that, in some cases, we can remove spurious dependencies in the approximating distribution. However, this result is surprising, since it holds regardless of the actual values of the potentials in P_Φ. And so, we can imagine a network where there are very strong interactions between A and C and between B and D in P_Φ, and yet the variational approximation with a network structure of figure 11.18c will not capture these dependencies. This is a consequence of using I-projections. Had we used an M-projection that minimizes $\boldsymbol{D}(P_\Phi \| Q)$, then we would have represented the dependencies between A and C; see exercise 11.30. ■

These two examples suggest that we can use the fixed-point characterization to refine an initial approximating network by factorizing its factors into a product of, possibly smaller, factors and potentials from P_Φ. We now consider the general theory of such factorizations and then discuss its implications.

We start with a simple definition and a proposition that form the basis of the simplifications we consider.

Definition 11.8

interface

*Let $\mathcal{H}$ be a Markov network structure and let $\boldsymbol{X}, \boldsymbol{Y} \subseteq \mathcal{X}$. We define the $\boldsymbol{Y}$-*interface *of $\boldsymbol{X}$, denoted Interface$_\mathcal{H}(\boldsymbol{X}; \boldsymbol{Y})$, to be the minimal subset of $\boldsymbol{X}$ such that* $\text{sep}_\mathcal{H}(\boldsymbol{X}; \boldsymbol{Y} \mid \text{Interface}_\mathcal{H}(\boldsymbol{X}; \boldsymbol{Y}))$. ■

That is, the $\boldsymbol{Y}$-interface of $\boldsymbol{X}$ is the subset of $\boldsymbol{X}$ that suffices to separate it from $\boldsymbol{Y}$.

Example 11.16

The $\{A, D\}$-interface of $\{A, B\}$ in $\mathcal{H}_{P_\Phi}$ of figure 11.18 is $\{A, B\}$, since neither A is separated from $\{A, D\}$ given B, nor is B is separated from $\{A, D\}$ given A. In $\mathcal{H}_{\mathcal{Q}_1}$, we have that B is separated from $\{A, D\}$ given A, so that Interface$_{\mathcal{H}_{\mathcal{Q}_1}}(\{A, B\}; \{A, D\})$ is $\{A\}$. The same holds in $\mathcal{H}_{\mathcal{Q}_3}$. In $\mathcal{H}_{\mathcal{Q}_2}$, we have that, again, neither A nor B suffices to separate the other from $\{A, D\}$, and hence, Interface$_{\mathcal{H}_{\mathcal{Q}_2}}(\{A, B\}; \{A, D\}) = \{A, B\}$. ■

The definition of interface can be used to reduce the scope of the conditional expectations in the fixed-point equations:

Proposition 11.6

If $\mathcal{H}$ is an I-map of $Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_j \psi_j$ and ϕ is a potential with scope $\boldsymbol{U}_\phi$. Then,

$$\boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\phi \mid \boldsymbol{c}_j] = \boldsymbol{E}_{\boldsymbol{U}_\phi \sim Q}[\phi \mid \boldsymbol{c}_j \langle \text{Interface}_\mathcal{H}(\boldsymbol{C}_j; \boldsymbol{U}_\phi) \rangle].$$

The proof follows immediately from the definition of conditional independence.

This proposition provides a principled approach for reformulating terms on the right-hand side of the fixed-point equation.

☞ **We can use this simplification result to define a two-phase strategy for designing approximation. First, we define a "rough" outline for approximation by defining Q over factors with a fairly large scope. We use this outline to obtain a set of update equations, as implied by equation (11.62) on Q. We then derive a finer-grained representation by factorizing each of these factors using proposition 11.6. This process results in a finer-grained approximation that is provably equivalent to the one with which we started.**

Theorem 11.13 **(Factorization)** *Let $\mathcal{Q}$ be an approximating family defined in terms of factors $\{\psi_j(\boldsymbol{C}_k)\}$, which induce a Markov network structure $\mathcal{H}_{\mathcal{Q}}$. Let $Q \in \mathcal{Q}$ be a stationary point of the energy functional $F[\tilde{P}_\Phi, Q]$ subject to the given factorization. Then, factors in Q are factorized as*

$$\psi_j(\boldsymbol{C}_j) = \prod_{\phi \in \Phi_j} \phi \prod_{\boldsymbol{D}_l \in \mathcal{D}_j} \psi_{j,l}(\boldsymbol{D}_l), \tag{11.65}$$

where

$$\Phi_j = \{\phi \in \Phi : \mathit{Scope}[\phi] \subseteq \boldsymbol{C}_j\}$$

and

$$\mathcal{D}_j = \{\mathit{Interface}_{\mathcal{H}_{\mathcal{Q}}}(\boldsymbol{C}_j; \boldsymbol{X}) : \boldsymbol{X} \in \{\mathit{Scope}[\phi] : \phi \in \Phi - \Phi_j\} \cup \{\mathit{Scope}[\psi_k] : k \neq j\}\}.$$

This theorem states that ψ_j can be written as the product of two sets of factors. The first set contains factors in the original distribution P_Φ whose scope is a subset of the scope of ψ_j. The factors in the second set are the interfaces of ψ_j with other factors that appear in the update equation. These include factors in P_Φ that are partially "covered" by the scope of ψ_k, and other factors in Q. The set $\mathcal{D}_k$ defines the set of interfaces between ψ_k and these factors.

To gain a better understanding of this theorem, let us consider various approximations in two concrete examples. The first example serves to demonstrate the ease with which this theorem allows us to determine the form of the factorization of Q.

Example 11.17 *Let us return to example 11.14. In example 11.16, we have already shown the interfaces of $\{A, B\}$ with $\{A, D\}$ in $\mathcal{H}_1$. This analysis, together with theorem 11.13, directly imply the reduced factorization of example 11.14 In particular, for $\psi_1(\{A, B\})$, we have that Φ_1 contains only the factor $\phi(\{A, B\})$ in P_Φ, which therefore constitutes the first term in the factorization of equation (11.65). The second set of terms in the equation corresponds to the interfaces of $\{A, B\}$ with other factors in both $\mathcal{H}_{P_\Phi}$ and in $\mathcal{H}_{\mathcal{Q}_1}$. We get two such interfaces: one with scope $\{A\}$ from the factor $\phi(\{A, D\})$ in P_Φ, and one with scope $\{B\}$ from the factor $\phi(\{B, C\})$.*

Assume that we add the edge A—C, as in figure 11.18d. Now, $\mathit{Interface}_{\mathcal{H}_{\mathcal{Q}_3}}(\{A, B\}; \{B, C\})$ is the entire set $\{A, B\}$, since B no longer separates C from A. Thus, in this case, the second set of terms in the factorization of ψ also contains a new pairwise interaction factor $\psi_{1,\{A,B\}}$. As a consequence, the pairwise interaction of A, B is no longer the same in Q and in P_Φ. This result is somewhat counterintuitive: In the simpler network $\mathcal{H}_{\mathcal{Q}_1}$, which contained no factors allowing any interaction between the A, B pair and the C, D pair, the A, B interaction was the same in P_Φ and

in Q. But if we enrich our approximation (presumably allowing a better fit via the introduction of the A, C factor), the pairwise interaction term does change.

Finally, $\mathcal{H}_{\mathcal{Q}_2}$ does not contain an $\{A, B\}$ factor. Here, $\Phi_j = \emptyset$ for both factors in $\mathcal{H}_{\mathcal{Q}_2}$, and each $\mathcal{D}_j$ consists solely of singleton scopes; for example, $\text{Interface}_{\mathcal{H}_{\mathcal{Q}_2}}(\{A, C\}; \{A, D\}) = \{A\}$. ■

Our second example serves to illustrate the two-phase strategy described earlier, where we first select a "rough" approximation containing a few large factors and then use the theorem to refine them.

Example 11.18

Consider again our running example of the 4×4 grid. Suppose we select an approximation where each factor consists of the variables in a single row in the grid. Thus, for example, $\boldsymbol{C}_1 = \{A_{1,1}, \ldots, A_{1,4}\}$. Note that this approximation is not the one shown in figure 11.17a, since the structure in our approximation here is a full clique over each row. We now apply theorem 11.13. What is the factorization of $\boldsymbol{C}_1$? First, we search for factors in Φ_1. We see that the factors $\phi(A_{1,1}, A_{1,2})$, $\phi(A_{1,2}, A_{1,3})$, and $\phi(A_{1,3}, A_{1,4})$ have a scope that is a subset of $\boldsymbol{C}_1$. Next, we consider the interfaces between $\boldsymbol{C}_1$ and other factors in P_Φ and Q. For example, the interface with $\phi(A_{1,1}, A_{2,1})$ is $\{A_{1,1}\}$. Similarly, $\{A_{1,2}\}$, $\{A_{1,3}\}$, and $\{A_{1,4}\}$ are interfaces with other factors in P_Φ. It is easy to convince ourselves that these are the only non-empty interfaces in $\mathcal{I}_1$. Thus, by applying theorem 11.13, we get the following factorization:

$$\begin{aligned}\psi_1(A_{1,1}, \ldots, A_{1,4}) &= \phi(A_{1,1}, A_{1,2}) \cdot \phi(A_{1,2}, A_{1,3}) \cdot \phi(A_{1,3}, A_{1,4}) \\ &\quad \psi_{1,1}(A_{1,1}) \cdot \psi_{1,2}(A_{1,2}) \cdot \psi_{1,3}(A_{1,3}) \cdot \psi_{1,4}(A_{1,4}).\end{aligned}$$

We conclude that, once we decide that the approximation should decouple the rows in the group, we might as well work with an approximation where we keep all original potentials along each row and introduce univariate potentials only to capture interactions along columns. Additional potentials, such as a potential between $A_{1,1}$ and $A_{1,3}$, would not improve the approximation. Thus, while we started with an approximation containing full cliques on each of the rows, we ended up with an approximation whose structure is that of figure 11.17a, and where we have only the original factors and new factors over single variables.

We can work directly with this new factorized form of Q, ignoring our original factorization entirely. More precisely, we define Q' to be

$$\begin{aligned}Q'(\mathcal{X}) &= \phi(A_{1,1}, A_{1,2}) \cdot \phi(A_{1,2}, A_{1,3}) \cdot \phi(A_{1,3}, A_{1,4}) \\ &\quad \cdots \\ &\quad \phi(A_{4,1}, A_{4,2}) \cdot \phi(A_{4,2}, A_{4,3}) \cdot \phi(A_{4,3}, A_{4,4}) \\ &\quad \psi_{1,1}(A_{1,1}) \cdot \ \cdot \psi_{4,4}(A_{4,4}).\end{aligned}$$

In this new form, we fix the value of all the pairwise potentials, and so we have to define an update rule only for the new singleton potentials. For example, consider the fixed-point equation for $\psi_{1,1}(A_{1,1})$. Applying theorem 11.12 we get that

$$\begin{aligned}&\ln \psi_{1,1}(a_{1,1}) \propto \\ &\quad + \boldsymbol{E}_{Q'}[\ln \phi(A_{1,1}, A_{2,1}) \mid a_{1,1}] + \boldsymbol{E}_{Q'}[\ln \phi(A_{1,2}, A_{2,2}) \mid a_{1,1}] \\ &\quad + \boldsymbol{E}_{Q'}[\ln \phi(A_{1,3}, A_{2,3}) \mid a_{1,1}] + \boldsymbol{E}_{Q'}[\ln \phi(A_{1,4}, A_{2,4}) \mid a_{1,1}] \\ &\quad - \boldsymbol{E}_{Q'}[\ln \psi_{1,2}(A_{1,2}) \mid a_{1,1}] - \boldsymbol{E}_{Q'}[\ln \psi_{1,3}(A_{1,3}) \mid a_{1,1}] - \boldsymbol{E}_{Q'}[\ln \psi_{1,4}(A_{1,4}) \mid a_{1,1}]\end{aligned}$$

where we have exploited the fact that the terms involving factors such as $\phi(A_{1,1}, A_{1,2})$ appear in both P_Φ and Q, and so cancel out of the equation. Note that to compute terms such as $\mathbf{E}_{Q'}[\ln \phi(A_{1,2}, A_{2,2}) \mid a_{1,1}]$ we need to evaluate $Q'(A_{1,2}, A_{2,2} \mid a_{1,1}) = Q'(A_{1,2} \mid a_{1,1}) \cdot Q'(A_{2,2})$ (where we used the independencies in Q' to simplify the joint marginal). Note that $Q'(A_{2,2})$ does not change when we update factors in the first row, such as $\psi_{1,1}(A_{1,1})$. Thus, we can cache the computation of this marginal when updating the factors $\psi_{1,1}(A_{1,1}), \ldots, \psi_{1,4}(A_{1,4})$. When performing inference in a large model this can result in dramatic effect. ■

cluster mean field

This example is a special case of an approximation approach called *cluster mean field*. In this case, our initial approximation has the form

$$Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_j \psi_j(\boldsymbol{C}_j),$$

where the scopes $\boldsymbol{C}_1, \ldots, \boldsymbol{C}_K$ are partition of $\mathcal{X}$. That is, each pair of factors have disjoint scopes, and each variable in $\mathcal{X}$ appears in one factor. This approximation resembles the mean field approximation, except that it is clusters, rather than individual variables, that are marginally independent. We can now apply theorem 11.13 to refine the approximation. Because the factors are all disjoint, there are no chains of influence, and so the interfaces take a particularly simple form:

Proposition 11.7

Let $Q(\mathcal{X}) = \frac{1}{Z_Q} \prod_j \psi_j(\boldsymbol{C}_j)$ be a cluster mean field approximation to a set of factors P_Φ, and let ψ_j be a factor of Q. Then, the set $\mathcal{D}_j$ of theorem 11.13 can be written as

$$\mathcal{D}_j = \{\boldsymbol{C}_j \cap \mathit{Scope}[\phi] : \phi \in \Phi - \Phi_j\} - \{\emptyset\}.$$

The proof follows directly from the independence properties in Q, and is left as an exercise (exercise 11.31).

In words, this result states that the interfaces of a cluster are simply the places where the cluster scope intersects potentials in Φ that are not fully contained in the cluster. In our grid example, when we choose the clusters to be the individual columns, the interfaces are the intersections with the row potentials, which are precisely the singleton variables that we discussed in example 11.18.

We conclude this discussion with a slightly more elaborate example, demonstrating again the strength of this result:

Example 11.19

Consider again our 4×4 grid, and the "comb" approximation whose structure is shown in figure 11.19a. In this structure, we have a fully connected clique over each of the columns, and a "backbone" connecting the columns to each other. Consider again the factorization of the potential over $\boldsymbol{C}_1 = \{A_{1,1}, \ldots, A_{4,1}\}$. As in the previous example, the first term in the new factorization contains the pairwise factors $\phi(A_{1,1}, A_{2,1})$, $\phi(A_{2,1}, A_{3,1})$, and $\phi(A_{3,1}, A_{4,1})$. The second set of terms contains the interfaces with other factors in P_Φ and Q. Due to the structure of the approximation, the Q interfaces introduce only singleton potentials. The factors in P_Φ, however, are more interesting. Consider, for example, the factor $\phi(A_{4,1}, A_{4,2})$. The interface of $\boldsymbol{C}_1$ with $\{A_{4,1}, A_{4,2}\}$ is $A_{1,1}, A_{4,1}$ — the variable $A_{4,1}$ separates $\boldsymbol{C}_1$ from itself, and the variable $A_{1,1}$ from $A_{4,2}$. Now, consider a factor $\phi(A_{2,3}, A_{3,3})$; in this case, the interface is simply $A_{1,1}$, which separates the first column from both of these variables. Continuing this argument, it follows that all other factors in

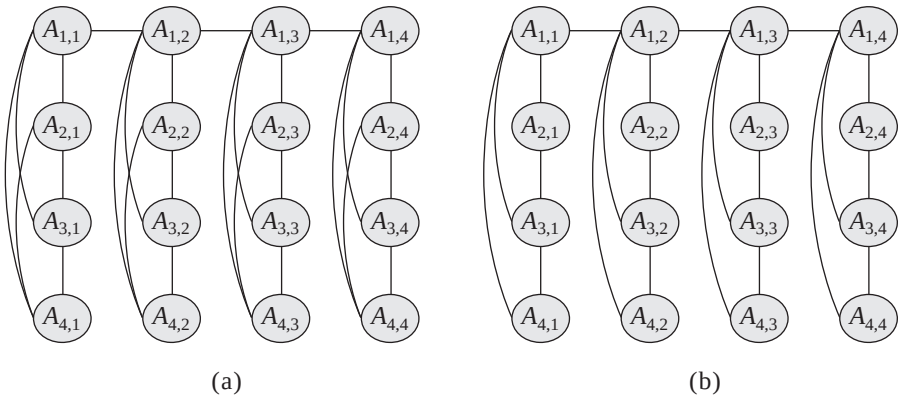


Figure 11.19 Simplification of approximating structure in cluster mean field. (a) Example of an approximating structure we can use in a variational approximation of a 4×4 grid network. (b) Simplification of that network using theorem 11.13.

P_Φ induce an interface containing a variable at the head of the column and (possibly) another variable in the column. Thus, we can eliminate any (new) pairwise interaction terms between any other pair of variables. For a general $n \times n$ grid, this reduces the overall number of (new) pairwise potentials from $n \cdot \binom{n}{2}$ to $n \times (n-1)$. ■

11.5.2.5 Selecting the Approximation

In general, both the quality and the computational complexity of the variational approximation depend on the structure of P_Φ and the structure of the approximating family $\mathcal{Q}$. There are several guiding intuitions. First, we want to be able to perform efficient inference in the approximating network. In example 11.18, the approximating structure was a chain of variables, where we can perform inference in linear time (as a function of the number of variables in the chain). In general, we often select our network so that the resulting factorization leads to a tractable network (that is, one of low tree-width).

It is important to note, however, that the structure of the original distribution is not the only aspect in determining the complexity of inference in Q. We also need to take into account factors that correspond to the interfaces of the cluster. In our grid example, these interfaces involved a single variable at time, and so they did not add to the network complexity. However, in more complex networks, these factors can have a significant effect.

Another consideration besides computational complexity is the quality of our approximation. Intuitively, we should design $\mathcal{Q}$ so as to preserve the strong dependencies in P_Φ. By preserving such dependencies we maintain the main effects in the distribution we want to apply.

These intuitions provide some guidelines in choosing the approximating distribution. However, these choices are far from an exact science at this stage. The theory we described here allows to automate two parts of the process: defining the form of the approximation given some initial rough set of (disjoint or overlapping) clusters; and defining the fixed-point iterations to

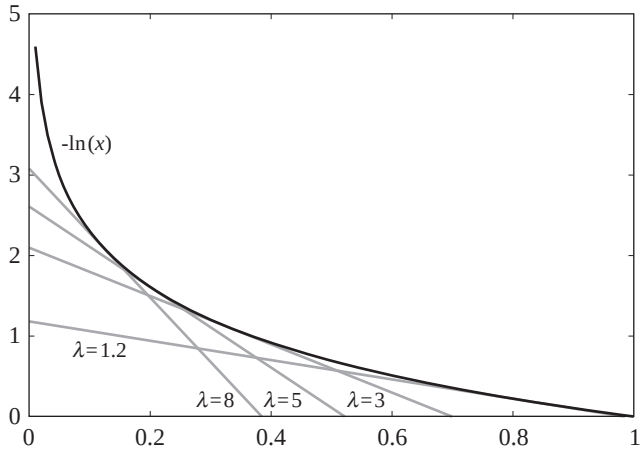


Figure 11.20 Illustration of the variational bound $-\ln(x) \geq -\lambda x + \ln(\lambda) + 1$**.** For any x, the bound holds for all λ, and is tight for some value of λ.

optimize such an approximation. The current tools do not provide for an automated way for determining what are reasonable sets of clusters to achieve a desired degree of approximation.

11.5.3 Local Variational Methods ⋆

variational method

lower bound

The general method that we used throughout this chapter is an instance of a general class of methods known as *variational methods*. In this class of methods, we take a complex objective function $f_{\text{obj}}(\boldsymbol{x})$, and lower or upper bound it using a parameterized family of functions $\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\lambda})$. Focusing, for concreteness, on the case of a *lower bound*, this family has the property that $f_{\text{obj}}(\boldsymbol{x}) \geq \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\lambda})$ for any value of $\boldsymbol{\lambda}$, and that, for any $\boldsymbol{x}$, the bound is tight for some value of $\boldsymbol{\lambda}$ (a different one for every $\boldsymbol{x}$).

variational lower bound

As an example, we can show the *variational lower bound*:

Lemma 11.2

For any choice of λ *and* x

$$-\ln(x) \geq -\lambda x + \ln(\lambda) + 1,$$

and, for any x*, this bound is tight for some value of* λ*.*

Proof Consider the tangent of $\ln(x)$ at the point x_0

$$f_{\text{obj}}(x : x_0) = \ln(x_0) + (x - x_0)\frac{1}{x_0} = \frac{x}{x_0} + \ln(x_0) - 1.$$

Since $\ln(x)$ is a concave function, it is upper bounded by each of its tangents. And so, $-\ln(x) \geq -f_{\text{obj}}(x : x_0)$ for any choice of x and x_0. Setting $x_0 = \lambda^{-1}$ leads to the desired result. ∎

convex duality

This result is illustrated in figure 11.20. It is a special case of a general result in the field of *convex duality*, which guarantees the existence of such bounds for a broad class of functions.

variational parameter

This lower bound allows to approximate a nonlinear function $-\ln(x)$ with a term that is linear in x. This simplification comes at the price of introducing a new *variational parameter* λ, whose value is undetermined. If we optimize λ exactly for each value of x, we obtain a tight lower bound, but a bound is obtained for any value of λ.

The techniques we have used in this chapter so far also fall into this category. Equation (11.5) shows that the energy functional is a lower bound on the log-partition function for any distribution Q. Thus, we can take f_{obj} to be the partition function, $\boldsymbol{x}$ to correspond to the parameters of the true distribution P_Φ, and $\boldsymbol{\lambda}$ to correspond to the parameters of the approximating distribution Q. Although the lower bound is tight when Q precisely represents P_Φ, for reasons of efficiency, we generally optimize Q in a restricted space that provides a bound, but not a tight one, on the log-partition function.

This general approach of introducing auxiliary variational parameters that help in simplifying a complex objective function appears in many other domains. While it is beyond our scope to introduce a general theory of variational methods, we now briefly describe one other application of variational methods that is relevant to probabilistic inference and does not fall directly within the scope of optimizing the energy functional. This application arises in the context of exact inference using an algorithm such as variable elimination. Here, we use variational bounds to avoid creating large factors that can lead to exponential complexity in the algorithm, giving rise to an approximate *variational variable elimination* algorithm. Such simplifications can be achieved in several ways; we describe two.

variational variable elimination

11.5.3.1 Variational Bounds

Consider, for example, the diamond network of figure 11.18a. Assume that we run variable elimination to sum out the variable B, which we assume for convenience is binary-valued. The elimination of B introduces a new factor:

$$\phi_B(A, C) = \sum_b \phi_1(A, b)\phi_2(b, C)$$

Coupling A and C in a single factor may be expensive, for example, if A and C have many values. In more complex networks, this type of coupling can induce complexity if an elimination step couples a larger set of variables, or if the local coupling leads to additional cost later in the computation, when we eliminate A or C.

As we now show, we can use a variational bound to avoid this coupling. Consider the following bound:

Proposition 11.8

If $0 \leq \lambda \leq 1$, *then*

$$\ln(1 + e^x) \geq \lambda x + \boldsymbol{H}(\lambda),$$

where $\boldsymbol{H}(\lambda) = -\lambda \ln \lambda - (1 - \lambda) \ln(1 - \lambda)$.

This bound implies that

$$1 + e^x \geq e^{\lambda x + \boldsymbol{H}(\lambda)}.$$

Why is this useful? Using some algebraic manipulation, we can bound each of the entries in our newly generated factor:

$$\begin{aligned}
\phi_B(a,c) &= \phi_1(b^0,a)\phi_2(b^0,c) + \phi_1(b^1,a)\phi_2(b^1,c) \\
&= \phi_1(b^0,a)\phi_2(b^0,c)\left[1 + \exp\left\{\ln \frac{\phi_1(b^1,a)\phi_2(b^1,c)}{\phi_1(b^0,a)\phi_2(b^0,c)}\right\}\right] \\
&\geq \phi_1(b^0,a)\phi_2(b^0,c)\exp\left\{\lambda_{a,c}\ln \frac{\phi_1(b^1,a)\phi_2(b^1,c)}{\phi_1(b^0,a)\phi_2(b^0,c)} + \boldsymbol{H}(\lambda_{a,c})\right\} \\
&= \left(\phi_1(b^0,a)^{1-\lambda_{a,c}}\phi_1(b^1,a)^{\lambda_{a,c}}\right) \cdot \\
&\quad \left(\phi_2(b^0,c)^{1-\lambda_{a,c}}\phi_2(b^1,c)^{\lambda_{a,c}}\right) \cdot e^{\boldsymbol{H}(\lambda_{a,c})}. \qquad (11.66)
\end{aligned}$$

Thus, we can replace a factor that couples A and C by a product of three terms: an expression involving only factors of A, an expression involving only factors of C, and the final factor $e^{\boldsymbol{H}(\lambda_{a,c})}$. However, all three terms also involve the variational parameter $\lambda_{a,c}$ and therefore also depend on both A and C. At this point, it is unclear what we gain from the transformation.

However, we can choose the same λ for all joint assignments to A, C. In doing so, we replace four variational parameters by a single parameter λ. This operation relaxes the bound, which is no longer tight. On the other hand, it also decouples A and C, leading to a product of terms none of which depends on both variables:

$$\left(\phi_1(b^0,a)^{1-\lambda}\phi_1(b^1,a)^{\lambda}\right) \cdot \left(\phi_2(b^0,c)^{1-\lambda}\phi_2(b^1,c)^{\lambda}\right) \cdot e^{\boldsymbol{H}(\lambda)} = \tilde{\phi}_1(a,\lambda)\tilde{\phi}_2(c,\lambda)e^{\boldsymbol{H}(\lambda)}. \quad (11.67)$$

Thus, if we use this approximation, we have effectively eliminated B without coupling A and C. As we saw in chapter 9, this type of simplification can circumvent the need for coupling yet more variables in later stages in variable elimination, potentially leading to significant savings.

It is interesting to observe how λ decouples the two factors. Each factor is replaced by a geometric average over the values of B. The variational parameter specifies the weight we assign to each of the two cases. Note that the original bound in equation (11.66) is tight; thus, if we pick the "right" variational parameter $\lambda_{a,c}$ for each assignment a, c, we reproduce the correct factor $\phi_B(A, C)$. However, these variational parameters are generally different for each assignment a, c, and hence, a single variational parameter cannot optimize all of the terms. Our choice of λ effectively determines the quality of our approximation for each of the terms $\phi_B(a, c)$. Thus, the overall quality of our approximation for a particular choice of λ depends on the importance of these different terms in the variable elimination computation as a whole.

Other variational approximations exploit specific parametric forms of CPDs in the network. For example, consider networks with sigmoid CPDs (see section 5.4.2). Recall that a logistic CPD $P(X \mid \boldsymbol{U})$ has the parametric form:

$$P(x^1 \mid \boldsymbol{u}) = \text{sigmoid}(\sum_i w_i u_i + w_0),$$

where $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$. The observation of X couples the parents $\boldsymbol{U}$. Can we decouple these parents using an approximation? Using proposition 11.8 we can find an upper bound:

$$\ln \text{sigmoid}(\sum_i w_i u_i + w_0) \leq \lambda\left(\sum_i w_i u_i + w_0\right) - \boldsymbol{H}(\lambda). \qquad (11.68)$$

Similarly to our earlier example, such an approximation allows us to replace a factor over several variables by a product of smaller factors. In this case, all the parents of X are decoupled by the approximate form.

11.5.3.2 Variational Variable Elimination

How do we use this approximation in the course of inference? Note that equation (11.67) provides a lower bound to $\phi_B(a, c)$ for every value of a, c. Assume that, in the course of running variable elimination, rather than generating $\phi_B(A, C)$, we introduce the expression in equation (11.67) and continue the variable elimination process with these decoupled factors. From a graph-theoretic perspective, the result of this approximation when applied to a variable B is analogous to the effect of conditioning on B, as described in section 9.5.3: it removes from the graph B and all its adjacent edges. However, unlike conditioning, we do not enumerate and perform inference for all values of B (of course, at the cost of obtaining an approximate result).

More generally, in an execution of variable elimination, there may be some set of elimination steps that create large factors that couple many variables. This variational approximation can allow us to avoid this coupling. Like conditioning (see section 9.5.4.1), we can perform such an approximation step not only at the very beginning, but also in a way that is interleaved with variable elimination, allowing us to reuse computation. This class of algorithms is called *variational variable elimination*.

variational variable elimination

What is the result of this approximation? Each of the entries in the approximated factor is replaced with a lower bound; thus, each entry in every subsequent factor produced by the algorithm is also a lower bound to the original entry. If we proceed to eliminate all variables, either exactly or using additional variational approximation steps for other intermediate factors, the outcome of this process is a lower bound to the partition function. If we do not eliminate all of the variables, the result is an approximate factor in which every entry is a lower bound to the original. Of course, once we renormalize the factor to produce a distribution, we can make no guarantees about the direction of the approximation for any given entry. Nevertheless, the resulting factor might be a reasonable approximation to the original.

The quality of our approximation depends on the choice of variational parameters introduced during the course of the variable elimination algorithm. How do we select them? One approach is simply to select the variational parameter at each step to optimize the quality of our approximation at that step. However, this high-level goal is not fully defined. For example, we can choose λ so as to make $\tilde{\phi}_1(a^1, \lambda)\tilde{\phi}_2(b^1, \lambda)e^{\boldsymbol{H}(\lambda)}$ as close as possible to $\phi_B(a^1, c^1)$; or, we can focus on a^1, c^0. The decision of where to focus our "approximation effort" depends on the impact of these components of the factor on the final outcome of the computation. Thus, a more correct approach is to identify the actual expression that we are trying to estimate — for example, the partition function — and to try to maximize our bound to that expression. In our simple example, we can write down the partition function as a function of the variational parameter λ introduced when eliminating B:

$$\tilde{Z}(\lambda) = e^{\boldsymbol{H}(\lambda)} \sum_a \sum_c \phi_3(a, d)\phi_4(c, d) \sum_d \tilde{\phi}_1(a, \lambda)\tilde{\phi}_2(c, \lambda).$$

This expression is a function of λ; we can then try to identify the best bound by maximizing $\max_\lambda \tilde{Z}(\lambda)$, say, using gradient ascent or another numerical optimization method.

However, as we discussed, in most cases we would use several approximate elimination steps within the variable elimination algorithm. In our example, the elimination of D also couples A and C, a situation we may wish to avoid. Thus, we could apply the same type of variational bound to the internal summation:

$$\phi_D(A, C) = \sum_d \tilde{\phi}_1(a, \lambda)\tilde{\phi}_2(c, \lambda),$$

giving rise to the bound $\tilde{\phi}_3(a, \lambda')\tilde{\phi}_4(c, \lambda')e^{\boldsymbol{H}(\lambda')}$. The resulting approximate partition function now has the form

$$\tilde{Z}(\lambda, \lambda') = e^{\boldsymbol{H}(\lambda)}e^{\boldsymbol{H}(\lambda')} \sum_a \tilde{\phi}_1(a, \lambda)\tilde{\phi}_3(a, \lambda') \sum_c \tilde{\phi}_2(c, \lambda)\tilde{\phi}_4(c, \lambda').$$

We can now maximize $\max_{\lambda,\lambda'} \tilde{Z}(\lambda, \lambda')$; the higher the value we find, the better our approximation to the true partition function.

In general, our approximate partition function will be a function $\tilde{Z}(\boldsymbol{\lambda})$, where $\boldsymbol{\lambda}$ is the vector of all variational parameters λ produced during the different approximation steps in the algorithm. One approach is to reformulate the variable elimination algorithm so that it produces factors that are not purely numerical, but rather symbolic expressions in the variables $\boldsymbol{\lambda}$; see exercise 11.32. Given the multivariate function $\tilde{Z}(\boldsymbol{\lambda})$, we can optimize it numerically to produce the best possible bound. A similar principle applies when we use the variational bound for sigmoid CPDs; see exercise 11.36 for an example. While we only sketch the basic idea here, this approach can be used as the basis for an algorithm that interleaves variational approximation steps with exact elimination steps to form a variational variable elimination algorithm.

11.6 Summary and Discussion

In this chapter, we have described a general class of methods for performing approximate inference in a distribution P_Φ defined by a graphical model. These methods all attempt to construct a representation Q within some approximation class $\mathcal{Q}$ that best approximates P_Φ. The key issue that must be tackled in this task is the construction of such an approximation without performing inference on the (intractable) P_Φ. The methods that we described all take a similar approach of using the I-projection framework, whereby we minimize the KL-divergence $\boldsymbol{I}\boldsymbol{D}(Q\|P_\Phi)$; these methods all reformulate this problem as one of maximizing the energy functional.

The different methods that we described all follow a similar template. We optimize the free energy, or an approximation thereof, over a class of representations $\mathcal{Q}$. We provided an optimization-based view for four different methods, each of which makes a particular choice regarding the objective and the constraints. We now recap these choices and their repercussions.

Clique tree calibration optimizes the factored energy functional, which is exact for clique trees. The optimization is performed over the space of calibrated clique potentials. For clique trees, any set of calibrated clique potentials must arise from a real distribution, and so this space is precisely the marginal polytope. As a consequence, both our objective and our constraint space are exact, so our solution represents the exact posterior.

Cluster-graph (loopy) belief propagation optimizes the factored energy functional, which is approximate for loopy graphs. The optimization is performed over the space of locally consistent

pseudo-marginals, which is a relaxation of the marginal polytope. Thus, both the objective and the constraint space are approximate.

Expectation propagation over clique trees optimizes the factored energy functional, which is the exact objective in this case. However, the constraints define a space of pseudo-marginals that are not even entirely consistent — only their moments are required to match. Thus, the constraint space here is a relaxation of our constraints, even when the structure is a tree. Expectation propagation over cluster graphs adds, on top of these, all of the approximations induced by belief propagation.

Finally, structured variational methods optimize the exact factored energy functional, for a class of distributions that is (generally) less expressive than necessary to encode P_Φ. As a consequence, the constraint space is actually a tightening of our original constraint space. Because the objective function is exact, the result of this optimization provides a lower bound on the value of the exact optimization problem. As we will see, such bounds can play an important role in the context of learning.

In the approaches we discussed, the optimization method was based on the method of Lagrange multipliers. This derivation gave rise to a series of fixed-point equations for the solution, where one variable is defined in terms of the others. As we showed, an iterative solution for these fixed-point equations gives rise to a suite of message passing algorithms that generalize the clique-tree message passing algorithms of chapter 10.

This general framework opens the door to the development of many other approaches. **Each of the methods that we described here involves a design choice in three dimensions: the objective function that we aim to optimize, the space of (pseudo-)distributions over which we perform our optimization, and the algorithm that we choose to use in order to perform the optimization. Although these three decisions affect each other, these dimensions are sufficiently independent that we can improve each one separately.** Some recent work takes exactly this approach. For example, we have already seen some work that focuses on better approximations to the energy functional; other work (see section 11.7) focuses on identifying constraints over the space of pseudo-marginals that make it a tighter relaxation to the (exact) marginal polytope; yet other work aims to find better (for example, more convergent) algorithms for a general class of optimization problems.

Many of these improvements aimed to address a fundamental problem arising in these algorithms: the possible lack of convergence to a stable solution. The issue of convergence is one on which some progress has been made. Some recently developed methods have better convergence properties in practice, and others are even guaranteed to converge. There are also theoretical analyses that can help determine when the algorithms converge.

A second key question, and one on which relatively little progress has been made, is the quality of the approximation. There is very little work that provides guarantees on the error made in the answers (for example, individual marginals) by any of these methods. As a consequence, there is almost no work that provides help in choosing a low-error approximation for a particular problem. Thus, the problem of applying these methods in practice is still a combination of manual tuning on one hand, and luck on the other. The development of automated methods that can help guide the selection of an approximating class for a particular problem is an important direction for future work.

11.7 Relevant Literature

Variational approximation methods are applicable in a wide variety of settings, including quantum mechanics (Sakurai 1985), statistical mechanics (Parisi 1988), neural networks (Elfadel 1995), and statistics (Rustagi 1976). The literature on the use of these approximation methods for inference problems in graphical models is heavily influenced by ideas in statistical mechanics, although the connections are beyond the scope of this book.

The rules for belief propagation were developed in Pearl's (1986b; 1988) analysis of inference in singly connected networks. In his book, Pearl noted that these propagation rules lead to wrong answers in multiconnected networks. Interest in "loopy" inference on pairwise networks was raised in several publications (Frey 1998; MacKay and Neal 1996; Weiss 1996) that reported good empirical results when running Pearl's algorithm on networks with loops. The most impressive success was in the error-correcting code scheme known as turbocodes (Berrou et al. 1993; McEliece et al. 1995). These decoding algorithms were shown to be equivalent to belief propagation in a network with loops (Kschischang and Frey 1998; McEliece et al. 1998; Frey and MacKay 1997). Researchers using these ad hoc algorithms reported that although they did not compute the correct posteriors, they often did compute the correct MAP assignment.

These empirical successes led to examination of both the reasons for the success of such methods and evaluated their performance in other domains. Weiss (2000) showed that when a network has a single loop, an analytic relationship exists between the belief computed and the true marginals, and it used that relationship to characterize network topologies for which the MAP solution is provably correct. Weiss and Freeman (2001a) then showed that if loopy belief propagation converges in a linear Gaussian network, then it computes correct posterior means; see section 14.7 for more references on results for the Gaussian case. These initial analytic results were supplemented by promising empirical results (Murphy et al. 1999; Horn and McEliece 1997; Weiss 2001).

Several alternative formulations of loopy belief propagation appeared in the literature. These include factor graphs (Kschischang et al. 2001b), tree-based reparameterization (Wainwright et al. 2003a), and algebraic formulations (Aji and McEliece 2000). In parallel, several authors (Yedidia et al. 2000; Dechter et al. 2002) proposed extending the idea of belief propagation to more generalized cluster graphs (or region graphs). Welling (2004) examined methods for automatically choosing the regions for this approximation.

A major advance in our understanding of this general class of approximation algorithms came with the analysis of Yedidia et al. (2000, 2005), showing that these methods can be posed as attempting to maximize an approximate energy functional. This result connected the algorithmic developments in the field with literature on free-energy approximations developed in statistical mechanics (Bethe 1935; Kikuchi 1951). This insight is the basis for our discussion of the Bethe energy functional in section 11.3.6. This result also provided a connection between belief propagation algorithms and other approximation procedures, such as structured variational methods. In addition, it led to the development of new types of approximate inference procedures. These include direct optimization of the Bethe energy functional (Welling and Teh 2001) using gradient-based methods, as well as provably convergent "double loop" variants of belief propagation (Yuille 2002; Heskes et al. 2003). Another class of algorithms combined exact inference on subtrees of the cluster graph within cluster-graph belief propagation (Wainwright et al. 2001, 2002a). This combination leads to faster convergence and introduces new directions

for characterizing the errors of the approximation. Wainwright and Jordan (2003) review the connections between belief propagation and optimization of the energy functional.

An active area of research is improving and analyzing the convergence properties of generalized belief propagation algorithms. In practice, the techniques most important for improving convergence are damping (Murphy et al. 1999; Heskes 2002) and message scheduling algorithms, such as the tree-based reparameterization algorithm of (Wainwright et al. 2003a) or the residual belief propagation algorithm of Elidan et al. (2006). On the theoretical side, key directions include the identification of a set of sufficient conditions for the existence of unique local maxima of the Bethe energy functional (Pakzad and Anantharam 2002; Heskes 2002), as well as conditions that ensure a unique convergence point of belief propagation (Tatikonda and Jordan 2002; Ihler et al. 2003; Heskes 2004; Ihler et al. 2005; Mooij and Kappen 2007). Our discussion in section 11.3.4 is based on that of Mooij and Kappen (2007). A related trajectory attempts to estimate the error of the approximate marginal probabilities; see, for example, Leisink and Kappen (2003); Ihler (2007).

A different generalizatiohn of belief-propagation algorithms develops variants of the energy functional that have desired properties. For example, if the energy functional is convex, then it is guaranteed to have a single maximum. An initial convexified free-energy functional was introduced in Wainwright et al. (2002b). This functional has the additional property of providing an upper bound on the partition function. Recently, there has been significant work that provides a more detailed characterization of convex energy functionals (Wainwright and Jordan 2003; Heskes 2006; Wainwright and Jordan 2004; Sontag and Jaakkola 2007). Although the convexity of energy functional implies a unique maximum, it does not generally guarantee that a message passing algorithm will converge. Recent alternative algorithms provide guaranteed convergence for such energy functionals (Heskes 2006; Globerson and Jaakkola 2007a; Hazan and Shashua 2008).

A recent extension is the combination of belief propagation with particle-based methods. The basic idea is to use particle sampling to perform the basic belief propagation steps (Sudderth et al. 2003; Isard 2003). This combination allows us to use cluster-graph belief propagation on networks with continuous non-Gaussian variables, which appear in applications in vision and signal processing; see also section 14.7.

The idea of factored messages first appeared in several contexts (Dechter and Rish 1997; Dechter 1997; Boyen and Koller 1998b; Murphy and Weiss 2001). Similar ideas that involve projection of messages onto "simple" representations (such as Gaussians) are common in the control and tracking literature (Bar-Shalom 1992). The generalization of these ideas in the form of expectation propagation was introduced by Minka (2001b). The connection between expectation propagation and the Bethe energy functional was explored by (Minka 2001b; Heskes and Zoeter 2002; Heskes et al. 2005). The connection between expectation propagation and generalized belief propagation in the case of discrete variables is explored by Welling et al. (2005).

One of the early applications of variational methods to probabilistic methods was the use of mean field approximation for Boltzmann machines (Peterson and Anderson 1987), an approach then widely adopted in computer vision (see, for example, Li (2001)). This methodology was introduced into the field of directed graphical models by Saul et al. (1996), following ideas that appeared in the context of Helmholtz machines (Hinton et al. 1995).

The mean field approximation cannot capture strong dependencies in the posterior distribution. Saul and Jordan (1996) suggested to circumvent this problem by using structured variational methods. These ideas were extended for different forms of approximating distributions

and target networks. Ghahramani and Jordan (1997) used independent hidden Markov chains to approximate factorial HMMs, a specific form of a dynamic Bayesian network. Barber and Wiegerinck (1998) use a Boltzmann machine approximation. Wiegerinck (2000) uses Markov networks and cluster trees as approximate distribution. Xing et al. (2003) describe cluster mean field and suggest efficient implementations. Geiger et al. (2006) further extend Wiegerinck's procedure and introduce efficient propagation schemes to maximize parameters in the approximating distribution.

The idea of using a mixture of mean field approximation was developed by Jaakkola and Jordan (1998). These ideas were extended to general networks with auxiliary variables by El-Hay and Friedman (2001).

Jaakkola and Jordan (1996b,a) introduced local variational approximations to compute both upper bounds and lower bounds of the log-likelihood function (that is, of $\log Z$). They demonstrated these methods by a large scale study of inference in the QMR-DT network (Jaakkola and Jordan 1999), where they show that variational methods are more effective than particle based methods. Additional extension on these methods appeared in Ng and Jordan (2000).

Tutorials discussing the use of structured and local methods appear in Jordan et al. (1998); Jaakkola (2001).

11.8 Exercises

Exercise 11.1

entropy

Show that the derivative of the *entropy* is:

$$\frac{\partial}{\partial Q(\boldsymbol{x})} \boldsymbol{H}_Q(\boldsymbol{X}) = -\ln Q(\boldsymbol{x}) - 1.$$

Exercise 11.2⋆

local consistency polytope

Consider the set of locally consistent distributions, as in the *local consistency polytope* of equation (11.16).

a. For a cluster graph $\mathcal{U}$ that is a clique tree $\mathcal{T}$ (satisfying the running intersection property), show that the set of distributions satisfying these constraints is precisely the *marginal polytope* — the set of legal distributions Q that can be represented over $\mathcal{T}$.

marginal polytope

b. Show that, for a cluster graph that is not a tree, there are parameterizations that satisfy these constraints but are not marginals of any legal probability distribution.

Exercise 11.3⋆

In the text, we showed that CTree-Optimize has a unique global optimum. Show that it also has a unique fixed point.

Exercise 11.4⋆

Consider the network of figure 11.1c. We have shown that

$$P_{\mathcal{T}}(A, B, C, D) \propto P_\Phi(A, B, C, D) \frac{\mu_{3,4}[D]\mu_{1,4}[A]}{\beta_4(A, D)},$$

where $\mathcal{T}$ is the cluster tree we get if we remove $\boldsymbol{C}_4 = \{A, D\}$. Show how to use this result on the *residual* to bound the error in the estimation of marginals in this cluster graph.

cluster graph residual

Exercise 11.5⋆

Prove proposition 11.2.

Exercise 11.6

Compare the cluster graphs in figure 11.3 and figure 11.7.

a. Show that $\mathcal{U}_2$ and $\mathcal{U}_3$ are equivalent in the following sense: Any set of calibrated potentials of one can be transformed to a calibrated set of potentials of the other.
b. Show similar equivalence between $\mathcal{U}_1$ and $\mathcal{U}_3$.

Exercise 11.7

image segmentation

As we discussed in box 4.B, Markov networks are commonly used for image-processing tasks. We now consider the application of Markov networks to foreground-background *image segmentation*. Here, we do not have a predetermined set of classes, each with its own model. Rather, for each pixel, we have a binary-valued variable X_i, where x_i^1 means that X_i is a foreground pixel, and x_i^0 a background pixel. In this case, we have no node potentials (say that features of an individual pixel cannot help distinguish foreground from background). The network structure is a grid, where for each pair of adjacent pixels i, j, we have an edge potential $\phi_{i,j}(X_i, X_j) = \alpha_{i,j}$ if $X_i = X_j$ and 1 otherwise. A large value of $\alpha_{i,j}$ makes it more likely that $X_i = X_j$ (that is, pixels i and j are assigned to the same segment), and a small value makes it less likely.

Using the standard Bethe cluster graph, compute the first message sent by loopy belief propagation from any variable to any one of its neighbors. What is wrong with this approach?

Exercise 11.8

Let X be a node with parents $\boldsymbol{U} = \{U_1, \ldots, U_k\}$, where $P(X \mid \boldsymbol{U})$ is a tree-CPD. Assume that we have a cluster consisting of X and $\boldsymbol{U}$. In this question, you will show how to exploit the structure of the tree-CPD to perform message passing more efficiently.

a. Consider a step where our cluster gets incoming messages about $U_1, \ldots, U_k$, and sends a message about X. Show how this step can be executed in time linear in the size of the tree-CPD of $P(X \mid \boldsymbol{U})$.
b. Now, consider the step where our clique gets incoming messages about $U_1, \ldots, U_{k-1}$ and X, and send a message about U_k. Show how this step can also be executed in time linear in the size of the tree-CPD.

Exercise 11.9

Recall that a pairwise Markov network consists of univariate potential $\phi_i[X_i]$ over each variable X_i, and in addition a pairwise potential $\phi_{(i,j)}[X_i, X_j]$ over some pairs of variables. In section 11.3.5.1 we showed a simple transformation of such a network to a cluster graph where we introduce a cluster for each potential. Write the update equations for cluster-graph belief propagation for such a network. Show that we can formulate inference as the direct propagation of messages between variables by implicitly combining messages through pairwise potentials.

Exercise 11.10⋆

Suppose we are given a set of factors $\Phi = \{\phi_1, \ldots, \phi_K\}$ over $\boldsymbol{X} = \{X_1, \ldots, X_n\}$. Our aim is to convert these factors into a pairwise Markov network by introducing new auxiliary variables $\boldsymbol{Y} = \{Y_1, \ldots, Y_k\}$ so that Y_j denotes a joint assignment to *Scope*$[\phi_j]$. Show how to construct a set of factors Φ' that is a pairwise Markov network over $\boldsymbol{X} \cup \boldsymbol{Y}$ such that $P_{\Phi'}(\boldsymbol{x}) = P_\Phi(\boldsymbol{x})$ for each assignment to $\boldsymbol{X}$.

Exercise 11.11⋆

BN cluster graph

In this question, we study how we can define a *cluster graph for a Bayesian network*.

a. Consider the following two schemes for converting a Bayesian network structure $\mathcal{G}$ to a cluster graph $\mathcal{U}$. For each of these two schemes, either show (by proving the necessary properties) that it produces a valid cluster graph for a general Bayesian network, or disprove this result by showing a counterexample.
   - **Scheme 1:** For each node X_i in $\mathcal{G}$, define a cluster $\boldsymbol{C}_i$ over X_i's family. Connect $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ if X_j is a parent of X_i in $\mathcal{G}$, and define the sepset to be the intersection of the clusters.

- **Scheme 2:** For each node X_i in $\mathcal{G}$, define a cluster $\boldsymbol{C}_i$ over X_i's family. Connect $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ if X_j is a parent of X_i in $\mathcal{G}$, and define the sepset to be the $\{X_j\}$.

b. Construct an alternative scheme to the ones proposed earlier that uses a minimum spanning tree algorithm to transform any Bayesian network into a valid cluster graph.

Exercise 11.12⋆

Suppose we want to use a gradient method to directly maximize $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ with respect to entries in $\boldsymbol{Q}$. For simplicity, assume that we are dealing with a pairwise network for both P_Φ and Q, and so the entries in $\boldsymbol{Q}$ are all univariate and pairwise potentials.

a. Derive $\frac{\partial \tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]}{\partial Q(x_i)}$ and $\frac{\partial \tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]}{\partial Q(x_i, y_i)}$ for the two types of potentials we have.

b. Recall that we cannot simply choose $\boldsymbol{Q}$ to maximize $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$. We need to ensure that every potential in $\boldsymbol{Q}$ is nonnegative and sums to 1. In addition, we need to maintain the marginal consistency between each pairwise potential and the associated univariate potential marginals in $\boldsymbol{Q}$. A standard solution is to write $\boldsymbol{Q}$ as a function of meta parameters η, so that the transformation from η to $\boldsymbol{Q}$ ensures the consistency. Suggest such a reparameterization, and derive the gradient of $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$ with respect to this reparameterization. (Hint: use the chain law of partial derivatives.)

Exercise 11.13⋆

Prove that if the damped updates of equation (11.14) converge, then they converge to a stationary point of $\tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}]$.

Exercise 11.14⋆

In this exercise you will prove theorem 11.7.

a. Start by defining the Lagrangian

$$\begin{aligned}\mathcal{J} &= \tilde{F}[\tilde{P}_\Phi, \boldsymbol{Q}] \\ &\quad - \sum_{i \in \mathcal{V}_\mathcal{R}} \lambda_r \left(\sum_{\boldsymbol{c}_r} \beta_r(\boldsymbol{c}_r) - 1 \right) \\ &\quad - \sum_{(r' \to r) \in \mathcal{E}_\mathcal{R}} \sum_{\boldsymbol{c}_r} \lambda_{r' \to r}[\boldsymbol{c}_r] \left(\sum_{\boldsymbol{c}_{r'} \sim \boldsymbol{c}_r} \beta_{r'}(\boldsymbol{c}_{r'}) - \beta_r(\boldsymbol{c}_r) \right),\end{aligned}$$

and show that any maximum point satisfies

$$\frac{\partial}{\partial \beta_r(\boldsymbol{c}_r)} \mathcal{J} = \kappa_r \ln \psi_r(\boldsymbol{c}_r) - \kappa_r \ln \beta_r(\boldsymbol{c}_r) - \kappa_r - \lambda_r - \sum_{r' \in \mathbf{Up}(r)} \lambda_{r' \to r}[\boldsymbol{c}_r] + \sum_{r' \in \mathbf{Down}(r)} \lambda_{r \to r'}[\boldsymbol{c}_{r'}],$$

where κ_r is the counting number of the region, as defined in equation (11.20).

b. Define $\delta_{r_u \to r_d}$ in terms of the Lagrange multipliers and show that your solution satisfies equation (11.37) and equation (11.36).

Exercise 11.15

Consider an $n \times n$ grid network. Construct a region-graph approximation for this network that would have a region r for each small square $A_{i,j}, A_{i,j+1}, A_{i+1,j}, A_{i+1,j+1}$. Describe the structure of the graph and the computation of the messages and beliefs.

Exercise 11.16

Prove theorem 11.6.

Exercise 11.17⋆

In this exercise, we will derive a message passing algorithm, called the *child-parent algorithm*, for Bethe region graphs. Although the messages in this algorithm are somewhat convoluted, they have the advantage of corresponding directly to the Lagrange multipliers for the region graph energy functional. Moreover, although somewhat opaque, the message passing algorithm is a very slight variation on the scheme used in standard belief propagation: the standard messages are simply raised to a power.

Consider the Bethe cluster graph of example 11.2, where we assume that all counting numbers are nonzero. Let $\rho_i = 1/\kappa_i$ and $\rho_r = 1/\kappa_r$. Starting from equation (11.34), derive the correctness of the following update rules for the messages and the beliefs. For all $r \in \mathbf{R}^+$:

$$\beta_r(\boldsymbol{C}_r) \leftarrow \left(\tilde{\psi}_r(\boldsymbol{C}_r) \prod_{X_i \in \boldsymbol{C}_r} \delta_{i \to r}(X_i) \right)^{\rho_r}. \tag{11.69}$$

$$\delta_{i \to r}(X_i) \leftarrow \left[\left(\prod_{r' \neg r} \delta_{i \to r'}(X_i) \right)^{\rho_i} \left(\sum_{\boldsymbol{C}_r - X_i} \psi_r(\boldsymbol{C}_r) \left(\prod_{X_j \in \boldsymbol{C}_r, j \neq i} \delta_{j \to r} \right)^{\rho_r} \right) \right]^{-\frac{1}{\rho_i + \rho_r}}. \tag{11.70}$$

Exercise 11.18⋆

Show that the counting numbers defined by equation (11.26) are convex. (Hint: Show first the convexity of counting numbers obtained from this analysis for a tree-structured MRF.)

Exercise 11.19⋆

In this exercise, we will derive another message passing algorithm, called the *two-way algorithm*, for finding fixed points of region-based energy functionals; this algorithm allows for more general region graphs than in exercise 11.17. It uses two messages along each link $r \to r'$: one from r to r' and one from r' to r.

Consider a region-based free energy as in equation (11.27). For any region r, let $p_r = |\mathbf{Up}(r)|$ be the number of regions that are directly upward of r. Assume that for any top region (so that $p_r = 0$), we have that $\kappa_r = 1$. We now define $q_r = (1 - \kappa_r)/p_{region}$, taking $q_r = 1$ when $p_r = 0$ (so that $\kappa_r = 1$, as per our assumption). Assume furthermore that $q_r \neq 2$, and define $\beta_r = 1/(2 - q_r)$.

The following equalities define the messages and the potentials in this algorithm:

$$\tilde{\psi}_r(\boldsymbol{C}_r) = (\psi_r(\boldsymbol{C}_r))^{\kappa_r} \tag{11.71}$$

$$\tilde{\delta}^{up}_{r \to r'} = \tilde{\psi}_r(\boldsymbol{C}_r) \prod_{r'' \in \mathbf{Up}(r) - \{r'\}} \tilde{\delta}^{down}_{r'' \to r}(\boldsymbol{C}_r) \prod_{r'' \in \mathbf{Down}(r)} \tilde{\delta}^{up}_{r'' \to r}(\boldsymbol{C}_{r''}) \tag{11.72}$$
$$(\psi_r(\boldsymbol{C}_r))^{\kappa_r}$$

$$\tilde{\delta}^{down}_{r' \to r} = \sum_{\boldsymbol{C}_{r'} - \boldsymbol{C}_r} \tilde{\psi}_r(\boldsymbol{C}_r) \prod_{r'' \in \mathbf{Up}(r)} \tilde{\delta}^{down}_{r'' \to r}(\boldsymbol{C}_r) \prod_{r'' \in \mathbf{Down}(r) - \{r'\}} \tilde{\delta}^{up}_{r'' \to r}(\boldsymbol{C}_{r''}) \tag{11.73}$$

$$\delta^{up}_{r \to r'} = (\tilde{\delta}^{up}_{r \to r'}(\boldsymbol{C}_r))^{\beta_r} (\tilde{\delta}^{down}_{r' \to r}(\boldsymbol{C}_r))^{\beta_r - 1} \tag{11.74}$$

$$\delta^{down}_{r \to r'} = (\tilde{\delta}^{up}_{r \to r'}(\boldsymbol{C}_r))^{\beta_r - 1} (\tilde{\delta}^{down}_{r' \to r}(\boldsymbol{C}_r))^{\beta_r} \tag{11.75}$$

$$\beta_r(\boldsymbol{C}_r) = \tilde{\psi}_r(\boldsymbol{C}_r) \prod_{r' \in \mathbf{Up}(r)} \delta^{down}_{r' \to r}(\boldsymbol{C}_r) \prod_{r'' \in \mathbf{Down}(r)} \delta^{up}_{r'' \to r}(\boldsymbol{C}_{r''}). \tag{11.76}$$

Note that the messages $\tilde{\delta}^{up}_{r \to r'}$ (or $\tilde{\delta}^{down}_{r \to r'}$) are as we would expect: the message sent from r to r' is simply the product of r's initial potential with all of its incoming messages except the one from r' (and similarly for the message sent from r' to r). However, as we discussed in our derivation of the region graph algorithm,

this computation will double-count the information that arrived at r from r' via an indirect path. The final computation of the messages $\delta^{up}_{r\to r'}$ and $\delta^{down}_{r\to r'}$ is intended to correct for that double-counting.

In this exercise, you will show that the fixed points of equation (11.33) are precisely the same as the fixed points of the update equations equation (11.71)–equation (11.76).

a. We begin by defining the messages in terms of the beliefs and the Lagrange multipliers:

$$\tilde{\delta}^{up}_{r\to r'}(\boldsymbol{c}_r) = \exp\{\lambda_{r,r'}(\boldsymbol{c}_r)\} \tag{11.77}$$

$$\tilde{\delta}^{down}_{r\to r'}(\boldsymbol{c}_{r'}) = \beta_{r'}(\boldsymbol{c}_{r'})^{q_{r'}} \exp\{\lambda_{r,r'}(\boldsymbol{c}_{r'})\}. \tag{11.78}$$

Show that these messages satisfy the fixed-point equations in equation (11.33).

b. Show that

$$(\beta_r(\boldsymbol{c}_r))^{\kappa_r} \propto (\beta_r(\boldsymbol{c}_r))^{\kappa_r - 1} \tilde{\psi}_r(\boldsymbol{c}_r) \prod_{r'\in \mathbf{Up}(r)} \delta^{down}_{r'\to r}(\boldsymbol{C}_r) \prod_{r''\in \mathbf{Down}(r)} \delta^{up}_{r''\to r}(\boldsymbol{C}_{r''}). \tag{11.79}$$

Conclude that equation (11.76) holds.

c. Show that

$$\tilde{\delta}^{up}_{r\to r'}\delta^{down}_{r'\to r} = \beta_r = \delta^{up}_{r\to r'}\tilde{\delta}^{down}_{r'\to r}, \tag{11.80}$$

and that

$$\delta^{up}_{r\to r'}\delta^{down}_{r'\to r} = (\beta_r)^{q_r}. \tag{11.81}$$

Show that the only solution to these equations is given by equation (11.73) and equation (11.74).

d. Show by direct derivation that theorem 11.6 holds for the potentials defined by equation (11.71)–11.76. (Hint: consider separately those regions that have no parents, and recall our previous assumptions.)

Exercise 11.20

Consider the marginal probability over $\boldsymbol{C}_2 = \{A_{1,2}, A_{2,2}, A_{3,2}, A_{4,2}\}$ in the 4×4 grid network. Show that, if we assume general potentials, this marginal probability does not satisfy any conditional independencies.

Exercise 11.21⋆

Consider a clique tree $\mathcal{T}$ that is constructed from a run of variable elimination over a Gibbs distribution P'_Φ, as described in section 10.1.2. Let $\boldsymbol{S}_{i,j}$ be a sepset in the clique tree. Show that, for general potentials, P'_Φ does not satisfy any conditional independencies relative to the marginal distribution over $\boldsymbol{S}_{i,j}$. (Hint: consider the actual run of variable elimination that led to the construction of this sepset, and the effect on the graph of eliminating these variables.)

Exercise 11.22⋆

In section 10.3.3.2 we described an algorithm that uses a calibrated clique tree to compute a joint marginal over a set of variables that is not a subset of any clique in the tree. We can consider using the same procedure in a calibrated cluster graph $\mathcal{U}$. For example, consider a pair of variables X, Y that are not found together in any cluster in $\mathcal{U}$.

a. Define an algorithm that, given clusters $\boldsymbol{C}_1 \ni X$ and $\boldsymbol{C}_2 \ni Y$ and a path between them in the cluster graph computes a joint distribution $P(X, Y)$.

b. A cluster graph can contain more than one path between $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$. Provide an example showing that the answer returned by this algorithm can depend on the choice of path used.

Exercise 11.23⋆

Suppose we have a cluster $\mathcal{T}$, and we consider two approximation schemes for inference in this cluster tree.

a. Using expectation propagation with fully factored approximation to the messages (as in figure 11.13).

b. Using a cluster graph approach on the cluster graph $\mathcal{U}$ constructed in the following manner. Each cluster $\boldsymbol{C}_i \in \mathcal{T}$ appears in $\mathcal{U}$. In addition, for each variable X_k that appears in some sepset $\boldsymbol{S}_{i,j}$ in $\mathcal{U}$ we introduce a new cluster with scope $\{X_k\}$ and connect it to both $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$. (This construction is similar to the Bethe cluster graph, except that we use the clusters in $\mathcal{T}$ as the "big" clusters in the construction.)

Show that both approximations are equivalent in the sense that their respective energy functionals (and the constraints they satisfy) coincide.

Exercise 11.24

Prove lemma 11.1.

Exercise 11.25⋆

In this exercise, you will provide a simpler proof for a special case of theorem 11.9 and corollary 11.6. Assume that each $X_i \in \mathcal{X}$ is a binary-valued random variable, parameterized with a single parameter $q_i = Q(x_i^1)$.

a. By considering the derivative of $F[\tilde{P}_\Phi, Q]$ and using lemma 11.1, prove theorem 11.9 without using Lagrange multipliers.

b. Now, prove corollary 11.6.

Exercise 11.26⋆

In this exercise, we will prove theorem 11.11. The proof relies on the following proposition, which characterizes the derivatives of an expectation relative to a Gibbs distribution.

Proposition 11.9 *If $Q(\mathcal{X}) = \frac{1}{Z_Q}\prod_j \psi_j$, then, for any function f with scope $\boldsymbol{U}$,*

$$\frac{\partial}{\partial \psi_j(\boldsymbol{c}_j)} \boldsymbol{E}_Q[f(\boldsymbol{U})] = \frac{Q(\boldsymbol{c}_j)}{\psi_j(\boldsymbol{c}_j)} \left(\boldsymbol{E}_Q[f(\boldsymbol{U}) \mid \boldsymbol{c}_j] - \boldsymbol{E}_Q[f(\boldsymbol{U})]\right) + \boldsymbol{E}_Q\left[\frac{\partial}{\partial \psi_j(\boldsymbol{c}_j)} f(\boldsymbol{U})\right].$$

a. Prove proposition 11.9.

b. Apply this proposition to prove theorem 11.11.

Exercise 11.27⋆

Consider the structured variational approximation of equation (11.61). As we discussed, to execute this update we need to collect the expectation of several factors, and each of these requires that we compute expectations given different assignments to the factor of interest. Assuming we use a clique tree to perform our inference, show how we can use a dynamic programming algorithm to reuse computation so as to evaluate these updates more efficiently.

Exercise 11.28

factorial HMM

Consider the *factorial HMM* model shown in figure 6.3a. Find a variational update rule for the structured variational approximation that factorizes as a set of clusters corresponding to the individual chains, that is, a cluster $\{X_i^{(0)}, \ldots, X_i^{(T)}\}$ for each i. Make sure that you simplify your clusters as much as possible, as in section 11.5.2.3 and section 11.5.2.4.

Exercise 11.29⋆

Now, consider the DBN whose structure is a set of variables X_i that evolve over time, where, at each time point, the chains are correlated by a tree structure that is the same for all time slices. Let Pa_i be the parents of X_i in the tree. Such structures (or extensions along similar lines) arise in several applications, such as those involving evolution of DNA or protein sequences; here, the chains encode spatial correlations over the sequence, and the tree the evolutionary process of each letter (DNA base pair or protein amino acid), whether across species, as in the *phylogenetic HMM* of box 6.A, or within a family tree (as in box 3.B).

phylogenetic HMM

Consider an unrolled network over time $0, \ldots, T$, where the initial $X_i^{(0)}$ are all independent. Find update rules for the following cluster variational approximations. In both cases, make sure that you simplify your clusters as much as possible, as in section 11.5.2.3 and section 11.5.2.4.

a. Find an update rule for the cluster variational approximation that has a cluster for each chain; that is, a cluster $\{X_i^{(0)}, \ldots, X_i^{(T)}\}$ for each i.
b. Find an update rule for the cluster variational approximation that has a cluster for each time slice; that is, a cluster $\{X_1^{(t)}, \ldots, X_n^{(t)}\}$ for each t.

Exercise 11.30

Consider a distribution P_Φ that consists of the pairwise Markov network of figure 11.18a, and consider approximating it with distribution Q that is represented by a pairwise Markov network of figure 11.18c. Derive the potentials $\psi_1(A, C)$ and $\psi_1(B, D)$ that maximize $\boldsymbol{I}\!D(P_\Phi \| Q)$. If A and C are not independent in P_Φ, will they be independent in the Q with these potentials?

Exercise 11.31⋆

Prove proposition 11.7.

Exercise 11.32⋆⋆

Describe an algorithm that performs variational variable elimination with optimization of the variational parameters. Your algorithm should take as input a set of factors Φ, an elimination ordering $X_1, \ldots, X_n$, and a subset $X_{i_1}, \ldots, X_{i_k}$ of steps at which variational approximations should be performed. Your algorithm should use the technique of section 11.5.3.2 to avoid coupling the factors in the elimination step of each X_{i_j}, introducing a single variational parameter λ_j for each such step. The result of the variable elimination procedure should be a symbolic expression in the λ_j's. Explain precisely how to construct this symbolic expression and how to optimize it to select the optimal set of variational parameters $\lambda_1, \ldots, \lambda_k$.

Exercise 11.33⋆

Show that if $Q = \prod_i Q(X_i \mid \boldsymbol{U}_i)$, then

$$\frac{\partial}{\partial Q(x_i \mid \boldsymbol{u}_i)} \boldsymbol{E}_Q[f(\boldsymbol{Y})] = \frac{1}{Q(x_i \mid \boldsymbol{u}_i)} \boldsymbol{E}_Q[f(\boldsymbol{Y}) \mid x_i, \boldsymbol{u}_i].$$

Exercise 11.34⋆⋆

Develop a variational approximation using Bayesian networks. Assume that Q is represented by a Bayesian network of a given structure $\mathcal{G}$. Derive the fixed-point characterization of parameters that maximize the energy functional (that is, the analog of theorem 11.11) for this type of approximation. (Hint: use theorem 8.2 and exercise 11.33 in your derivation.)

Exercise 11.35⋆

Prove proposition 11.8.

Exercise 11.36⋆

In this exercise we consider inference in two-layered Bayesian networks with logistic CPDs. In these networks, all variables are binary. The variables $X_1, \ldots, X_n$ in the top layer are all independent of each

other. The variables $Y_1, \ldots, Y_m$ in the bottom layer depend on the variables in the top layer using a logistic CPD:

$$P(y_j^1 \mid x_1, \ldots, x_n) = \text{sigmoid}(\sum_i w_{i,j} x_i + w_{0,j}).$$

Suppose we observe some of the variables on the bottom layer, and we want to estimate the probability of the evidence.

a. Show that when computing the probability of evidence, we can remove variables Y_i that are not observed.
b. Given evidence $\boldsymbol{y}$, use the bound of equation (11.68) to write a variational upper bound of $P(\boldsymbol{y})$.
c. Develop a mean field approximation to lower-bound the probability of the same evidence.

Exercise 11.37⋆⋆

a. Show that the following function f_{obj} is convex:

$$f_{\text{obj}}(a_1, \ldots, a_n) = \ln \sum_i e^{a_i}.$$

b. Prove the following bound

$$f_{\text{obj}}(a_1, \ldots, a_n) \geq \sum_i \lambda_i a_i - \sum_i \lambda_i \ln \lambda_i,$$

for any choice of $\{\lambda_i\}$ so that $\sum_i \lambda_i = 1$ and $\lambda_i \geq 0$ for all i. (Hint: use the convexity of $f_{\text{obj}}()$.)

c. Write

$$\ln Z = f_{\text{obj}}(\{\ln \tilde{P}_\Phi(\xi)\}),$$

and use the preceding bound to write a lower bound for $\ln Z$. Use this analysis to provide an alternative derivation of the mean field approximation.

Exercise 11.38⋆⋆

mixture distribution

We now consider a very different class of representations that we can use in a structure variational approximation. Here, our distribution Q is a *mixture distribution*. More precisely, in addition to the variables $\mathcal{X}$ in P_Φ, we introduce a new variable T. We can now define an approximating family:

$$Q(\mathcal{X}) = \sum_t Q(t, \mathcal{X}) = \sum_t Q(t) Q(\mathcal{X} \mid t), \tag{11.82}$$

which is a mixture of different approximations $Q(\mathcal{X} \mid t)$. As one simple instantiation, we might have:

$$Q(\mathcal{X}, T) = Q(T) \prod_i Q(X_i \mid T), \tag{11.83}$$

where each mixture component, $Q(\mathcal{X} \mid t)$, is a mean field distribution.

a. Assuming that Q is structured as in equation (11.82), prove that

$$F[\tilde{P}_\Phi, Q] = \sum_t Q(t) F[\tilde{P}_\Phi, Q(\mathcal{X} \mid t)] + \mathbf{I}_Q(T; \mathcal{X}), \tag{11.84}$$

where $\mathbf{I}_Q(T; \mathcal{X})$ is the mutual information between T and $\mathcal{X}$.

This result quantifies the gains that we can obtain by using a mixture distribution. The first term is the weighted average of the energy functional of the mixture components; this term is bounded by the best of the components in isolation. The improvement over using specific components is captured by the second term, which measures the extent to which T influences the distribution over $\mathcal{X}$. If the components represent identical distributions, then this term is zero, and, as expected, the value of the energy functional is identical to the one obtained using the component representation. By contrast, if the components are different, then T is informative on $\mathcal{X}$, and the lower bound provided by Q can be better than that of each of the components. We note that the "best scenario" for improvement is when the component terms $F[\tilde{P}_\Phi, Q(\mathcal{X} \mid t)]$ are similar, and yet $\mathbf{I}_Q(T; \mathcal{X})$ is large. In other words, the best approximation is obtained when each of the components provides a good approximation, but by using a different distribution.

We now use this formulation of the energy functional to produce an update rule for this variational approximation. This derivation uses a local variational approximation.

b. Let $\{\lambda(\xi, t) : \xi \in Val(\mathcal{X}), t \in Val(T)\}$ be a family of constants. Use lemma 11.2 to show that:

$$\mathbf{I}_Q(T; \mathcal{X}) \geq \mathbf{H}_Q(T) - \sum_{\xi,t} \lambda(\xi, t) Q(t) + \mathbf{E}_Q[\ln \lambda(\mathcal{X}, T)] + 1.$$

Show also that this bound is tight if we choose a different value of $\lambda(\xi, t)$ for each possible combination of ξ, t.

c. Suppose that we use a factorized form for the variational parameters:

$$\lambda(\xi, t) = \lambda(t) \prod_i \lambda(x_i \mid t).$$

Show that

$$F[\tilde{P}_\Phi, Q] \geq \sum_t Q(t) F[\tilde{P}_\Phi, Q(\mathcal{X} \mid t)] + \mathbf{H}_Q(T) - \mathbf{E}_Q\left[\tilde{\lambda}(T)\right] + \tag{11.85}$$

$$\mathbf{E}_Q[\lambda(T)] + \sum_i \mathbf{E}_Q[\lambda(X_i \mid T)], \tag{11.86}$$

where $\tilde{\lambda}(t) = \sum_\xi \lambda(\xi, t)$.

d. Now, assume concretely that our approximation Q has the form of equation (11.83), and that all of the X_i variables are binary-valued. Use your lower bound in equation (11.85) as an approximation to the free energy, and provide an update rule for all of the parameters in Q: $P(t)$ and $P(x_i^1 \mid t)$.

e. Analyze the computational complexity of applying these fixed-point equations.

12 *Particle-Based Approximate Inference*

In the previous chapter, we discussed one class of approximate inference methods. The techniques used in that chapter gave rise to algorithms that were very similar in flavor to the factor-manipulation methods underlying exact inference. In this chapter, we discuss a very different class of methods, ones that can be roughly classified as *particle-based methods*. In these methods, we approximate the joint distribution as a set of instantiations to all or some of the variables in the network. These instantiations, often called *particles*, are designed to provide a good representation of the overall probability distribution.

particle

Particle-based methods can be roughly characterized along two axes. On one axis, approaches vary on the process by which particles are generated. There is a wide variety of possible processes. At one extreme, we can generate particles using some deterministic process. At another, we can sample particles from some distribution. Within each category, there are many possible variations.

On the other axis, techniques use different notions of particles. Most simply, we can consider *full particles* — complete assignments to all of the network variables $\mathcal{X}$. The disadvantage of this approach is that each particle covers only a very small part of the space. A more effective notion is that of a *collapsed particle*. A collapsed particle specifies an assignment $\boldsymbol{w}$ only to some subset of the variables $\boldsymbol{W}$, associating with it the conditional distribution $P(\mathcal{X} \mid \boldsymbol{w})$ or some "summary" of it.

full particle

collapsed particle

The general framework for most of the discussion in this chapter is as follows. Consider some distribution $P(\mathcal{X})$, and assume we want to estimate the probability of some event $\boldsymbol{Y} = \boldsymbol{y}$ relative to P, for some $\boldsymbol{Y} \subseteq \mathcal{X}$ and $\boldsymbol{y} \in Val(\boldsymbol{Y})$. More generally, we might want to estimate the expectation of some function $f(\mathcal{X})$ relative to P; this task is a generalization, since we can choose $f(\xi) = \boldsymbol{1}\{\xi\langle\boldsymbol{Y}\rangle = \boldsymbol{y}\}$, where we recall that $\xi\langle\boldsymbol{Y}\rangle$ is the assignment in ξ to the variables $\boldsymbol{Y}$. We approximate this expectation by generating a set of M particles, estimating the value of the function or its expectation relative to each of the generated particles, and then aggregating the results.

For most of this chapter, we focus on methods that generate particles using random sampling: In section 12.1, we consider the simplest possible method, which simply generates samples from the original network. In section 12.2, we present a significantly improved method that generates samples from a distribution that is closer to the posterior distribution. In section 12.3, we discuss a method based on Markov chains that defines a sampling process that, as it converges, generates samples from distributions arbitrarily close to the posterior. In section 12.5, we consider a very different type of method, one that generates particles deterministically by

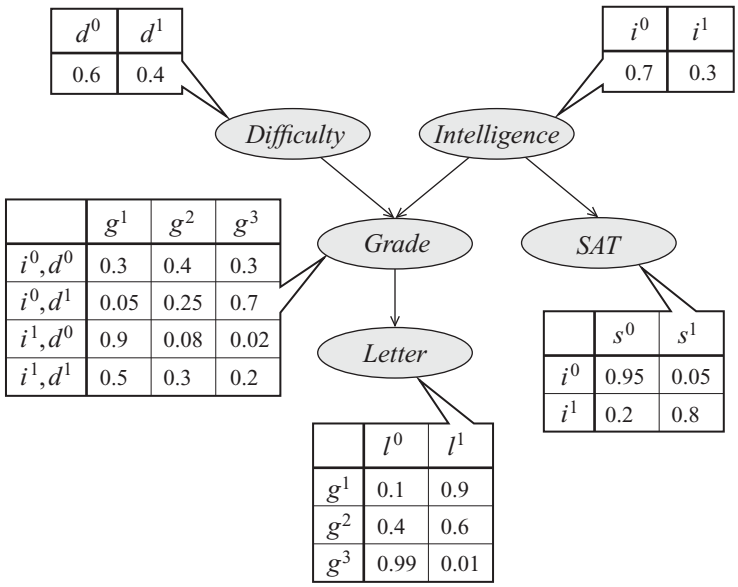


Figure 12.1 The Student **network $\mathcal{B}^{student}$ revisited**

searching for high-probability instantiations in the joint distribution. Finally, in section 12.4, we extend these methods to the case of collapsed particles. We note that, unlike our discussion of exact inference, some of the methods presented in this chapter — forward sampling and likelihood weighting — apply (at least in their simple form) only to Bayesian networks, and not to Markov networks or chain graphs.

12.1 Forward Sampling

forward sampling

The simplest approach to the generation of particles is *forward sampling*. Here, we generate random samples $\xi[1], \ldots, \xi[M]$ from the distribution $P(\mathcal{X})$. We first show how we can easily generate particles from $P_{\mathcal{B}}(\mathcal{X})$ by sampling from a Bayesian network. We then analyze the number of particles needed in order to get a good approximation of the expectation of some target function f. We finally discuss the difficulties in generating samples from the posterior $P_{\mathcal{B}}(\mathcal{X} \mid \boldsymbol{e})$. We note that, in undirected models, even generating a sample from the prior distribution is a difficult task.

12.1.1 Sampling from a Bayesian Network

Sampling from a Bayesian network is a very simple process.

Example 12.1

Consider the Student *network, shown again in figure 12.1. We begin by sampling D with the appropriate (unconditional) distribution; that is, we figuratively toss a coin that lands heads (d^1) 40 percent of the time and tails (d^0) the remaining 60 percent. Let us assume that the coin landed*

Algorithm 12.1 Forward Sampling in a Bayesian network

```
Procedure Forward-Sample (
    B    // Bayesian network over X
)
1   Let X_1, ..., X_n be a topological ordering of X
2   for i = 1, ..., n
3     u_i ← x⟨Pa_{X_i}⟩    // Assignment to Pa_{X_i} in x_1, ..., x_{i-1}
4     Sample x_i from P(X_i | u_i)
5   return (x_1, ..., x_n)
```

heads, so that we pick the value d^1 for D. Similarly, we sample I from its distribution; say that the result is i^0. Given those, we know the right distribution from which to sample G: $P(G \mid i^0, d^1)$, as defined by G's CPD; we therefore pick G to be g^1 with probability 0.05, g^2 with probability 0.25, and g^3 with probability 0.7. The process continues similarly for S and L. ■

As shown in algorithm 12.1, we sample the nodes in some order consistent with the partial order of the BN, so that by the time we sample a node we have values for all of its parents. We can then sample from the distribution defined by the CPD and by the chosen values for the node's parents. Note that the algorithm requires that we have the ability to sample from the distributions underlying our CPD. Such sampling is straightforward in the discrete case (see box 12.A), but subtler when dealing with continuous measures (see section 14.5.1).

Box 12.A — Skill: Sampling from a Discrete Distribution. *How do we generate a sample from a distribution? For a uniform distribution, we can use any pseudo-random number generator on our machine. Other distributions require more thought, and much work has been devoted in statistics to the problem of sampling from a variety of parametric distributions. Most obviously, consider a multinomial distribution $P(X)$ for $Val(X) = \{x^1, \ldots, x^k\}$, which is defined by parameters $\theta_1, \ldots, \theta_k$. This process can be done quite simply as follows: We generate a sample s uniformly from the interval $[0, 1]$. We then partition the interval into k subintervals: $[0, \theta_1), [\theta_1, \theta_1 + \theta_2), \ldots$; that is, the ith interval is $[\sum_{j=1}^{i-1} \theta_j, \sum_{j=1}^{i} \theta_j)$. If s is in the ith interval, then the sampled value is x^i. We can determine the interval for s using binary search in time $O(\log k)$.*

This approach gives us a general-purpose solution for generating samples from the CPD of any discrete-valued variable: given a parent assignment $\boldsymbol{u}$, we can always generate the full conditional distribution $P(X \mid \boldsymbol{u})$ and sample from it. (Of course, more efficient methods may exist if X has a large value space or a CPD that requires an expensive computation.) As we discuss in section 14.5.1, the problem of sampling from continuous CPDs is considerably more complex.

convergence bound

Using basic *convergence bounds* (see appendix A.2), we know that from a set of particles $\mathcal{D} = \{\xi[1], \ldots, \xi[M]\}$ generated via this sampling process, we can estimate the expectation of

any function f as:

$$\hat{\boldsymbol{E}}_{\mathcal{D}}(f) = \frac{1}{M}\sum_{m=1}^{M} f(\xi[m]). \tag{12.1}$$

In the case where our task is to compute $P(\boldsymbol{y})$, this estimate is simply the fraction of particles where we have seen the event $\boldsymbol{y}$:

$$\hat{P}_{\mathcal{D}}(\boldsymbol{y}) = \frac{1}{M}\sum_{m=1}^{M} \boldsymbol{I}\{\boldsymbol{y}[m] = \boldsymbol{y}\}, \tag{12.2}$$

where we use $\boldsymbol{y}[m]$ to denote $\xi[m]\langle \boldsymbol{Y}\rangle$ — the assignment to $\boldsymbol{Y}$ in the particle $\xi[m]$. For example, our estimate for the probability of an event such as i^1, l^0 (a smart student getting a bad letter) is the fraction of particles in which I got the value i^1 and L the value l^0. Note that the same set of particles can be used to estimate the probabilities of multiple events.

This sampling process requires one sampling operation for each random variable in the network. For each variable X, we need to index into the CPD using the current partial instantiation of the parents of X. Using an appropriate data structure, this indexing can be accomplishing in time $O(|\mathrm{Pa}_X|)$. The actual sampling process, as we discussed, requires time $O(\log |Val(X)|)$ (assuming appropriate preprocessing). Letting M be the total number of particles generated, $n = |\mathcal{X}|$, $p = \max_i |\mathrm{Pa}_{X_i}|$, and $d = \max_i |Val(X_i)|$, the overall cost is $O(Mnp \log d)$.

12.1.2 Analysis of Error

Of course, the quality of the estimate obtained depends heavily on the number of particles generated. We now analyze the question of the number of particles required to obtain certain performance guarantees. We focus on the analysis for the case where our goal is to estimate $P(\boldsymbol{y})$.

The techniques of appendix A.2 provide us with the necessary tools for this analysis. Consider the quality of our estimate for a particular event $\boldsymbol{Y} = \boldsymbol{y}$. We define a new random variable over the probability space of P, using the indicator function $\boldsymbol{I}\{\boldsymbol{Y} = \boldsymbol{y}\}$. This is a Bernoulli random variable, and hence our M particles in $\mathcal{D}$ define M independent Bernoulli trials, each with success probability $P(\boldsymbol{y})$.

Hoeffding bound

We can now apply the *Hoeffding bound* (theorem A.3) to show that this estimate is close to the truth with high probability:

$$P_{\mathcal{D}}(\hat{P}_{\mathcal{D}}(\boldsymbol{y}) \notin [P(\boldsymbol{y}) - \epsilon, P(\boldsymbol{y}) + \epsilon]) \leq 2e^{-2M\epsilon^2}. \tag{12.3}$$

This analysis provides us with an estimate of how many samples are required to achieve an estimate whose error is bounded by ϵ, with probability at least $1 - \delta$. Setting

$$2e^{-2M\epsilon^2} \leq \delta$$

sample size

estimator

and doing simple algebraic manipulations, we get that the required *sample size* to get an *estimator* with (ϵ, δ) reliability is:

$$M \geq \frac{\ln(2/\delta)}{2\epsilon^2}.$$

Chernoff bound
relative error

We can similarly apply the *Chernoff bound* (theorem A.4) to conclude that $\hat{P}_{\mathcal{D}}(\boldsymbol{y})$ is also within a *relative error* ϵ of the true value $P(\boldsymbol{y})$, with high probability. Specifically, we have that:

$$P_{\mathcal{D}}(\hat{P}_{\mathcal{D}}(\boldsymbol{y}) \notin P(\boldsymbol{y})(1 \pm \epsilon)) \leq 2e^{-MP(\boldsymbol{y})\epsilon^2/3}. \tag{12.4}$$

Note that in this analysis, unlike the one based on Hoeffding's bound, the error probability (the chance of getting an estimate that is more than ϵ away from the true value) depends on the actual target value $P(\boldsymbol{y})$. This dependence is not surprising for a relative error bound. Assume that we generate M samples, but we generate none where $\boldsymbol{y}[m] = \boldsymbol{y}$. Our estimate $\hat{P}_{\mathcal{D}}(\boldsymbol{y})$ is simply 0. However, if $P(\boldsymbol{y})$ is very small, it is fairly likely that we simply have not generated any samples where this event holds. In this case, our estimate of 0 is not going to be within any relative error of $P(\boldsymbol{y})$. Thus, for very small values of $P(\boldsymbol{y})$, we need many more samples in order to guarantee that our estimate is close with high probability.

Examining equation (12.4), we can see that, for a given ϵ, the number of samples needed to guarantee a certain error probability δ is:

$$M \geq 3\frac{\ln(2/\delta)}{P(\boldsymbol{y})\epsilon^2}. \tag{12.5}$$

Thus, the number of required samples grows inversely with the probability $P(\boldsymbol{y})$.

In summary, to guarantee an absolute error of ϵ with probability at least $1 - \delta$, we need a number of samples that grows logarithmically in $1/\delta$ and quadratically in $1/\epsilon$. To guarantee a relative error of ϵ with error probability at most δ, we need a number of samples that grows similarly in δ and ϵ, but that also grows linearly with $1/P(\boldsymbol{y})$. A significant problem with using this latter bound is that we do not know $P(\boldsymbol{y})$. (If we did, we would not have to estimate it.) Thus, we cannot determine how many samples we need in order to ensure a good estimate.

12.1.3 Conditional Probability Queries

So far, we have focused on the problem of estimating marginal probabilities, that is, probabilities of events $\boldsymbol{Y} = \boldsymbol{y}$ relative to the original joint distribution. In general, however, we are interested in conditional probabilities of the form $P(\boldsymbol{y} \mid \boldsymbol{E} = \boldsymbol{e})$. Unfortunately, it turns out that this estimation task is significantly harder.

rejection sampling

One approach to this task is simply to generate samples from the posterior probability $P(\mathcal{X} \mid \boldsymbol{e})$. We can do so by a process called *rejection sampling*: We generate samples $\boldsymbol{x}$ from $P(\boldsymbol{X})$, as in section 12.1.1. We then reject any sample that is not compatible with $\boldsymbol{e}$. The resulting samples are sampled from the posterior $P(\boldsymbol{X} \mid \boldsymbol{e})$. The results of the analysis in section 12.1.2 now apply unchanged.

The problem, of course, is that the number of unrejected particles can be quite small. In general, the expected number of particles that are not rejected from an original sample set of size M is $MP(\boldsymbol{e})$. For example, if $P(\boldsymbol{e}) = 0.001$, then even for $M = 10,000$ samples, the expected number of unrejected particles is 10. Conversely, to obtain at least M^* unrejected particles, we need to generate on average $M = M^*/P(\boldsymbol{e})$ samples from $P(\boldsymbol{X})$.

Unfortunately, in many applications, low-probability evidence is the rule rather than the exception. For example, in medical diagnosis, any set of symptoms typically has low probability. In general, as the number of observed variables $k = |\boldsymbol{E}|$ grows, the probability of the evidence usually decreases exponentially with k.

An alternative approach to the problem is to use a separate estimator for $P(\boldsymbol{e})$ and for $P(\boldsymbol{y}, \boldsymbol{e})$ and then compute the ratio. We can show that if we have estimators of low relative error for both of these quantities, then their ratio will also have a low relative error (exercise 12.2). Unfortunately, this approach only moves the problem from one place to the other. As we said, **the number of samples required to achieve a low relative error also grows linearly with $1/P(\boldsymbol{e})$. The number of samples required to get low *absolute* error does not grow with $P(\boldsymbol{e})$. However, it is not hard to verify (exercise 12.2) that a bound on the absolute error for $P(\boldsymbol{e})$ does not suffice to get any type of bound (relative or absolute) for the ratio $P(\boldsymbol{y}, \boldsymbol{e})/P(\boldsymbol{e})$.**

12.2 Likelihood Weighting and Importance Sampling

The rejection sampling process seems very wasteful in the way it handles evidence. We generate multiple samples that are inconsistent with our evidence and that are ultimately rejected without contributing to our estimator. In this section, we consider an approach that makes our samples more relevant to our evidence.

12.2.1 Likelihood Weighting: Intuition

Consider the network in example 12.1, and assume that our evidence is d^1, s^1. Our forward sampling process might begin by generating a value of d^0 for D. No matter how the sampling process proceeds, this sample will always be rejected as being incompatible with the evidence.

It seems much more sensible to simply force the samples to take on the appropriate values at observed nodes. That is, when we come to sampling a node X_i whose value has been observed, we simply set it to its observed value.

In general, however, this simple approach can generate incorrect results:

Example 12.2

Consider the network of example 12.1, and assume that our evidence is s^1 — a student who received a high SAT score. Using the naive process, we sample D and I from their prior distribution, set $S = s^1$, and then sample G and L appropriately. All of our samples will have $S = s^1$, as desired. However, the expected number of samples that have $I = i^1$ — an intelligent student — is 30 percent, the same as in the prior distribution. Thus, this approach fails to conclude that the posterior probability of i^1 is higher when we observe s^1. ■

The problem with this approach is that it fails to account for the fact that, in the standard forward sampling process, the node S is more likely to take the value s^1 when its parent I has the value i^1 than when I has the value i^0. In particular, consider an imaginary process where we run rejection sampling many times; samples where we generated the value $I = i^1$ would have generated $S = s^1$ in 80 percent of the samples, whereas samples where we generated the value $I = i^0$ would have generated $S = s^1$ in only 5 percent of the samples. To simulate this long-run behavior within a single sample, we should conclude that a sample where we have

Algorithm 12.2 Likelihood-weighted particle generation

```
Procedure LW-Sample (
    B,    // Bayesian network over X
    Z = z    // Event in the network
)
1   Let X_1, ..., X_n be a topological ordering of X
2   w ← 1
3   for i = 1, ..., n
4     u_i ← x⟨Pa_{X_i}⟩    // Assignment to Pa_{X_i} in x_1, ..., x_{i-1}
5     if X_i ∉ Z then
6       Sample x_i from P(X_i | u_i)
7     else
8       x_i ← z⟨X_i⟩    // Assignment to X_i in z
9       w ← w · P(x_i | u_i)    // Multiply weight by probability of desired value
10  return (x_1, ..., x_n), w
```

$I = i^1$ and force $S = s^1$ should be worth 80 percent of a sample, whereas one where we have $I = i^0$ and force $S = s^1$ should only be worth 5 percent of a sample.

When we have multiple observations and we want our sampling process to set all of them to their observed values, we need to consider the probability that each of the observation nodes, had it been sampled using the standard forward sampling process, would have resulted in the observed values. The sampling events for each node in forward sampling are independent, and hence the weight for each sample should be the product of the weights induced by each evidence node separately.

Example 12.3

Consider the same network, where our evidence set now consists of l^0, s^1. Assume that we sample $D = d^1$, $I = i^0$, set $S = s^1$, sample $G = g^2$, and set $L = l^0$. The probability that, given $I = i^0$, forward sampling would have generated $S = s^1$ is 0.05. The probability that, given $G = g^2$, forward sampling would have generated $L = l^0$ is 0.4. If we consider the standard forward sampling process, each of these events is the result of an independent coin toss. Hence, the probability that both would have occurred is simply the product of their probabilities. Thus, the weight required for this sample to compensate for the setting of the evidence is $0.05 \cdot 0.4 = 0.02$. ■

likelihood weighting

Generalizing this intuition results in an algorithm called *likelihood weighting (LW)*, shown in algorithm 12.2. The name indicates that the weights of different samples are derived from the likelihood of the evidence accumulated throughout the sampling process.

weighted particle

This process generates a *weighted particle*. We can now estimate a conditional probability $P(\boldsymbol{y} \mid \boldsymbol{e})$ by using LW-Sample M times to generate a set $\mathcal{D}$ of weighted particles $\langle \xi[1], w[1] \rangle, \ldots, \langle \xi[M], w[M] \rangle$. We then estimate:

$$\hat{P}_{\mathcal{D}}(\boldsymbol{y} \mid \boldsymbol{e}) = \frac{\sum_{m=1}^{M} w[m] \mathbb{1}\{\boldsymbol{y}[m] = \boldsymbol{y}\}}{\sum_{m=1}^{M} w[m]}. \tag{12.6}$$

estimator

This *estimator* is an obvious generalization of the one we used for unweighted particles in

equation (12.2). There, each particle had weight 1; hence, the terms in the numerator were unweighted, and the denominator, which is the sum of all the particle weights, was simply M. It is also important to note that, as in forward sampling, the same set of samples can be used to estimate the probability of any event $\boldsymbol{y}$.

Aside from some intuition, we have provided no formal justification for the correctness of LW as yet. It turns out that LW is a special case of a very general approach called *importance sampling*, which also provides us the basis for an analysis. We begin by providing a general description and analysis of importance sampling, and then reformulate LW as a special case of this framework.

12.2.2 Importance Sampling

importance sampling

target distribution

Let $\boldsymbol{X}$ be a variable (or set of variables) that takes on values in some space $Val(\boldsymbol{X})$. *Importance sampling* is a general approach for estimating the expectation of a function $f(\boldsymbol{x})$ relative to some distribution $P(\boldsymbol{X})$, typically called the *target distribution*. As we discussed, we can estimate this expectation by generating samples $\boldsymbol{x}[1], \ldots, \boldsymbol{x}[M]$ from P, and then estimating

$$\boldsymbol{E}_P[f] \approx \frac{1}{M} \sum_{m=1}^{M} f(\boldsymbol{x}[m]).$$

proposal distribution

In some cases, however, we might prefer to generate samples from a different distribution Q, known as the *proposal distribution* or *sampling distribution*. There are several reasons why we might wish to sample from a different distribution. Most importantly for our purposes, it might be impossible or computationally very expensive to generate samples from P. For example, P might be a posterior distribution for a Bayesian network, or even a prior distribution for a Markov network.

In this section, we discuss how we might obtain estimates of an expectation relative to P by generating samples from a different distribution Q. In general, the proposal distribution Q can be arbitrary; we require only that $Q(\boldsymbol{x}) > 0$ whenever $P(\boldsymbol{x}) > 0$, so that Q does not "ignore" any states that have nonzero probability relative to P. (More formally, the *support* of a distribution P is the set of points $\boldsymbol{x}$ for which $P(\boldsymbol{x}) > 0$; we require that the support of Q contain the support of P.) However, as we will see, the computational performance of this approach does depend strongly on the extent to which Q is similar to P.

support

12.2.2.1 Unnormalized Importance Sampling

If we generate samples from Q instead of P, we cannot simply average the f-value of the samples generated. We need to adjust our estimator to compensate for the incorrect sampling distribution. The most obvious way of adjusting our estimator is based on the observation that

$$\boldsymbol{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})] = \boldsymbol{E}_{Q(\boldsymbol{X})}\left[f(\boldsymbol{X})\frac{P(\boldsymbol{X})}{Q(\boldsymbol{X})}\right]. \tag{12.7}$$

This equality follows directly:[1]

$$\begin{aligned} \mathbf{E}_{Q(\boldsymbol{X})}\left[f(\boldsymbol{X})\frac{P(\boldsymbol{X})}{Q(\boldsymbol{X})}\right] &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x})f(\boldsymbol{x})\frac{P(\boldsymbol{x})}{Q(\boldsymbol{x})} \\ &= \sum_{\boldsymbol{x}} f(\boldsymbol{x})P(\boldsymbol{x}) \\ &= \mathbf{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})]. \end{aligned}$$

Based on this observation, we can use the standard estimator for expectations relative to Q. We generate a set of samples $\mathcal{D} = \{\boldsymbol{x}[1], \ldots, \boldsymbol{x}[M]\}$ from Q, and then estimate:

$$\hat{\mathbf{E}}_{\mathcal{D}}(f) = \frac{1}{M}\sum_{m=1}^{M} f(\boldsymbol{x}[m])\frac{P(\boldsymbol{x}[m])}{Q(\boldsymbol{x}[m])}. \tag{12.8}$$

unnormalized importance sampling estimator

We call this estimator the *unnormalized importance sampling* estimator; this method is also often called *unweighted* importance sampling (this terminology is confusing, inasmuch as the particles here are also associated with weights). The factor $P(\boldsymbol{x}[m])/Q(\boldsymbol{x}[m])$ can be viewed as a correction weight to the term $f(\boldsymbol{x}[m])$, which we would have used had Q been our target distribution. We use $w(\boldsymbol{x})$ to denote $P(\boldsymbol{x})/Q(\boldsymbol{x})$.

unbiased estimator

Our analysis immediately implies that this estimator is *unbiased*, that is, its mean for any data set is precisely the desired value:

Proposition 12.1

For data sets $\mathcal{D}$ sampled from Q, we have that:

$$\mathbf{E}_{\mathcal{D}}\left[\hat{\mathbf{E}}_{\mathcal{D}}(f)\right] = \mathbf{E}_{Q(\boldsymbol{X})}[f(\boldsymbol{X})w(\boldsymbol{X})] = \mathbf{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})].$$

We can also estimate the distribution of this estimator around its mean. Letting $\epsilon_{\mathcal{D}} = \hat{\mathbf{E}}_{\mathcal{D}}(f) - \mathbf{E}_P[f(\boldsymbol{x})]$, we have that, since $M \to \infty$:

$$\mathbf{E}_{\mathcal{D}}[\epsilon_{\mathcal{D}}] \sim \mathcal{N}\left(0; \sigma_Q^2/M\right),$$

where

$$\begin{aligned} \sigma_Q^2 &= \mathbf{E}_{Q(\boldsymbol{X})}\left[(f(\boldsymbol{X})w(\boldsymbol{X}))^2\right] - \mathbf{E}_{Q(\boldsymbol{X})}[(f(\boldsymbol{X})w(\boldsymbol{X}))]^2 \\ &= \mathbf{E}_{Q(\boldsymbol{X})}\left[(f(\boldsymbol{X})w(\boldsymbol{X}))^2\right] - (\mathbf{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})])^2. \end{aligned} \tag{12.9}$$

estimator variance

As we discussed in appendix A.2, the *variance* of this type of estimator — an average of M independent random samples from a distribution — decreases linearly with the number of samples. This point is important, since it allows us to provide a bound on the number of samples required to obtain a reliable estimate.

To understand the constant term in this expression, consider the (uninteresting) case where the function f is the constant function $f(\xi) \equiv 1$. In this case, equation (12.9) simplifies to:

$$\mathbf{E}_{Q(\boldsymbol{X})}\left[w(\boldsymbol{X})^2\right] - \mathbf{E}_{P(\boldsymbol{X})}[1] = \mathbf{E}_{Q(\boldsymbol{X})}\left[\left(\frac{P(\boldsymbol{X})}{Q(\boldsymbol{X})}\right)^2\right] - \left(\mathbf{E}_{Q(\boldsymbol{X})}\left[\frac{P(\boldsymbol{X})}{Q(\boldsymbol{X})}\right]\right)^2,$$

1. We present the proof in terms of discrete state spaces, but it holds equally for continuous state spaces.

which is simply the variance of the weighting function $P(x)/Q(x)$. Thus, the more different Q is from P, the higher the variance of this estimator. When f is an indicator function over part of the space, we obtain an identical expression restricted to the relevant subspace. In general, one can show that the lowest variance is achieved when

$$Q(\boldsymbol{X}) \propto |f(\boldsymbol{X})|P(\boldsymbol{X});$$

thus, for example, if f is an indicator function over part of the space, we want our sampling distribution to be P conditioned on the subspace.

Note that we should avoid cases where our sampling probability $Q(\boldsymbol{X}) \ll P(\boldsymbol{X})f(\boldsymbol{X})$ in any part of the space, since these cases can lead to very large or even infinite variance. Thus, care must be taken when using very skewed sampling distributions, to ensure that probabilities in Q are close to zero only when $P(\boldsymbol{X})f(\boldsymbol{X})$ is also very small.

12.2.2.2 Normalized Importance Sampling

One problem with the preceding discussion is that it assumes that P is known. A frequent situation, and one of the most common reasons why we must resort to sampling from a different distribution Q, is that P is known only up to a normalizing constant Z. Specifically, what we have access to is a function $\tilde{P}(\boldsymbol{X})$ such that $\tilde{P}$ is not a normalized distribution, but $\tilde{P}(\boldsymbol{X}) = ZP(\boldsymbol{X})$. For example, in a Bayesian network $\mathcal{B}$, we might have (for $\boldsymbol{X} = \mathcal{X}$) $P(\mathcal{X})$ be our posterior distribution $P_{\mathcal{B}}(\mathcal{X} \mid \boldsymbol{e})$, and $\tilde{P}(\mathcal{X})$ be the unnormalized distribution $P_{\mathcal{B}}(\mathcal{X}, \boldsymbol{e})$. In a Markov network, $P(\mathcal{X})$ might be $P_{\mathcal{H}}(\mathcal{X})$, and $\tilde{P}$ might be the unnormalized distribution obtained by multiplying together the clique potentials, but without normalizing by the partition function.

In this context, we cannot define the weights relative to P, so we define:

$$w(\boldsymbol{X}) = \frac{\tilde{P}(\boldsymbol{X})}{Q(\boldsymbol{X})}. \tag{12.10}$$

Unfortunately, with this definition of weights, the analysis justifying the use of equation (12.8) breaks down. However, we can use a slightly different estimator based on similar intuitions. As before, the weight $w(\boldsymbol{X})$ is a random variable. Its expected value is simply Z:

$$\mathbf{E}_{Q(\boldsymbol{X})}[w(\boldsymbol{X})] = \sum_{\boldsymbol{x}} Q(\boldsymbol{x}) \frac{\tilde{P}(\boldsymbol{x})}{Q(\boldsymbol{x})} = \sum_{\boldsymbol{x}} \tilde{P}(\boldsymbol{x}) = Z. \tag{12.11}$$

This quantity is the normalizing constant of the distribution $\tilde{P}$, which is itself often of considerable interest, as we will see in our discussion of learning algorithms.

We can now rewrite equation (12.7):

$$\begin{aligned}
\mathbf{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})] &= \sum_{\boldsymbol{x}} P(\boldsymbol{x}) f(\boldsymbol{x}) \\
&= \sum_{\boldsymbol{x}} Q(\boldsymbol{x}) f(\boldsymbol{x}) \frac{P(\boldsymbol{x})}{Q(\boldsymbol{x})} \\
&= \frac{1}{Z} \sum_{\boldsymbol{x}} Q(\boldsymbol{x}) f(\boldsymbol{x}) \frac{\tilde{P}(\boldsymbol{x})}{Q(\boldsymbol{x})} \\
&= \frac{1}{Z} \mathbf{E}_{Q(\boldsymbol{X})}[f(\boldsymbol{X}) w(\boldsymbol{X})] \\
&= \frac{\mathbf{E}_{Q(\boldsymbol{X})}[f(\boldsymbol{X}) w(\boldsymbol{X})]}{\mathbf{E}_{Q(\boldsymbol{X})}[w(\boldsymbol{X})]}. \qquad (12.12)
\end{aligned}$$

We can use an empirical estimator for both the numerator and denominator. Given M samples $\mathcal{D} = \{\boldsymbol{x}[1], \ldots, \boldsymbol{x}[M]\}$ from Q, we can estimate:

$$\hat{\mathbf{E}}_{\mathcal{D}}(f) = \frac{\sum_{m=1}^{M} f(\boldsymbol{x}[m]) w(\boldsymbol{x}[m])}{\sum_{m=1}^{M} w(\boldsymbol{x}[m])}. \qquad (12.13)$$

normalized importance sampling estimator

We call this estimator the *normalized importance sampling estimator*; it is also known as the *weighted* importance sampling estimator.

The normalized estimator involves a quotient, and it is therefore much more difficult to analyze theoretically. However, unlike the unnormalized estimator of equation (12.8), the normalized estimator is not unbiased. This bias is particularly immediate in the case $M = 1$. Here, the estimator reduces to:

$$\frac{f(\boldsymbol{x}[1]) w(\boldsymbol{x}[1])}{w(\boldsymbol{x}[1])} = f(\boldsymbol{x}[1]).$$

Because $\boldsymbol{x}[1]$ is sampled from Q, the mean of the estimator in this case is $\mathbf{E}_{Q(\boldsymbol{X})}[f(\boldsymbol{X})]$ rather than the desired $\mathbf{E}_{P(\boldsymbol{X})}[f(\boldsymbol{X})]$. Conversely, when M goes to infinity, we have that each of the numerators and denominators converges to the expected value, and our analysis of the expectation applies. In general, for finite M, the estimator is biased, and the bias goes down as $1/M$.

One can show that the variance of the importance sampling estimator with M data instances is approximately:

$$\mathbf{Var}_P\left[\hat{\mathbf{E}}_{\mathcal{D}}(f(\boldsymbol{X}))\right] \approx \frac{1}{M} \mathbf{Var}_P[f(\boldsymbol{X})](1 + \mathbf{Var}_Q[w(\boldsymbol{X})]), \qquad (12.14)$$

which also goes down as $1/M$. Theoretically, this variance and the variance of the unnormalized estimator (equation (12.8)) are incomparable, and each of them can be larger than the other. Indeed, it is possible to construct examples where each of them performs better than the other. In practice, however, the variance of the normalized estimator is typically lower than that of the unnormalized estimator. This reduction in variance often outweighs the bias term, so that the normalized estimator is often used in place of the unnormalized estimator, even in cases where P is known and we can sample from it effectively.

Note that equation (12.14) can be used to provide a rough estimate on the quality of a set of samples generated using normalized importance sampling. Assume that we were to estimate $\mathbf{E}_P[f]$ using a standard sampling method, where we generate M IID samples from $P(\boldsymbol{X})$. (Obviously, this is generally intractable, but it provides a useful benchmark for comparison.) This approach would result in a variance $\mathbf{Var}_P[f(\boldsymbol{X})]/M$. The ratio between these two variances is:

$$\frac{1}{1 + \mathbf{Var}_Q[w(\boldsymbol{x})]}.$$

Thus, we would expect M weighted samples generated by importance sampling to be "equivalent" to $M/(1 + \mathbf{Var}_Q[w(\boldsymbol{x})])$ samples generated by IID sampling from P. We can use this observation to define a rule of thumb for the *effective sample size* of a particular set $\mathcal{D}$ of M samples resulting from a particular run of importance sampling:

effective sample size

$$\begin{aligned} M_{\text{eff}} &= \frac{M}{1 + \mathbf{Var}[\mathcal{D}]} \\ \mathbf{Var}[\mathcal{D}] &= \sum_{m=1}^{M} w(\boldsymbol{x}[m])^2 - (\sum_{m=1}^{M} w(\boldsymbol{x}[m]))^2. \end{aligned} \tag{12.15}$$

This estimate can tell us whether we should continue generating additional samples.

12.2.3 Importance Sampling for Bayesian Networks

With this theoretical foundation, we can now describe the application of importance sampling to Bayesian networks. We begin by providing the proposal distribution most commonly used for Bayesian networks. This distribution Q uses the network structure and its CPDs to focus the sampling process on a particular part of the joint distribution — the one consistent with a particular event $\boldsymbol{Z} = \boldsymbol{z}$. We show several ways in which this construction can be applied to the Bayesian network inference task, dealing with various types of probability queries. Finally, we briefly discuss several other proposal distributions, which are somewhat more complicated to implement but may perform better in practice.

12.2.3.1 The Mutilated Network Proposal Distribution

Assume that we are interested in a particular event $\boldsymbol{Z} = \boldsymbol{z}$, either because we wish to estimate its probability, or because we have observed it as evidence. We wish to focus our sampling process on the parts of the joint that are consistent with this event. In this section, we define an importance sampling process that achieves this goal.

To gain some intuition, consider the network of figure 12.1 and assume that we are interested in a particular event concerning a student's grade: $G = g^2$. We wish to bias our sampling toward parts of the space where this event holds. It is easy to take this event into consideration when sampling L: we simply sample L from $P(L \mid g^2)$. However, it is considerably more difficult to account for G's influence on D, I, and S without doing inference in the network.

Our goal is to define a simple proposal distribution that allows for the efficient generation of particles. We therefore avoid the problem of accounting for the effect of the event on nondescendants; we define a proposal distribution that "sets" the value of a $Z \in \boldsymbol{Z}$ to take the

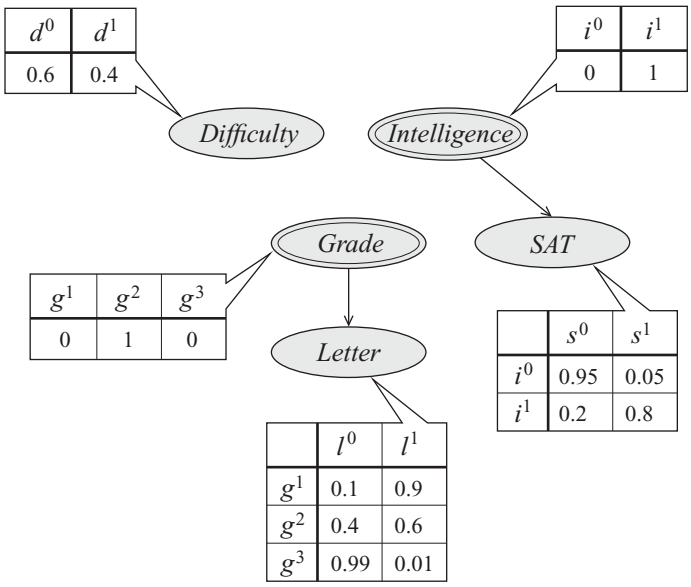


Figure 12.2 The mutilated network $\mathcal{B}^{student}_{I=i^1, G=g^2}$ used for likelihood weighting

prespecified value in a way that influences the sampling process for its descendants, but not for the other nodes in the network. The proposal distribution is most easily described in terms of a Bayesian network:

Definition 12.1 mutilated network

Let $\mathcal{B}$ be a network, and $\mathcal{Z}_1 = z_1, \ldots, Z_k = z_k$, abbreviated $\boldsymbol{Z} = \boldsymbol{z}$, an instantiation of variables. We define the mutilated network $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$ *as follows:*

- *Each node $Z_i \in \boldsymbol{Z}$ has no parents in $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$; the CPD of Z_i in $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$ gives probability 1 to $Z_i = z_i$ and probability 0 to all other values $z_i' \in Val(Z_i)$.*
- *The parents and CPDs of all other nodes $X \notin \boldsymbol{Z}$ are unchanged.* ■

For example, the network $\mathcal{B}^{student}_{I=i^1, G=g^2}$ is shown in figure 12.2. As we can see, the node G is decoupled from its parents, eliminating its dependence on them (the node I has no parents in the original network, so its parent set remains empty). Furthermore, both I and G have CPDs that are deterministic, ascribing probability 1 to their (respective) observed values.

Importance sampling with this proposal distribution is precisely equivalent to the LW algorithm shown in algorithm 12.2, with $\tilde{P}(\mathcal{X}) = P_{\mathcal{B}}(\mathcal{X}, \boldsymbol{z})$ and the proposal distribution Q induced by the mutilated network $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$. More formally, we can show the following proposition:

Proposition 12.2

Let ξ be a sample generated by algorithm 12.2 and w be its weight. Then the distribution over ξ is as defined by the network $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$, and

$$w(\xi) = \frac{P_{\mathcal{B}}(\xi)}{P_{\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}}(\xi)}.$$

The proof is not difficult and is left as an exercise (exercise 12.4). It is important to note, however, that the algorithm does not require the explicit construction of the mutilated network. It simply traverses the original network, using the process shown in algorithm 12.2.

As we now show, this proposal distribution can be used for estimating a variety of Bayesian network queries.

12.2.3.2 Unconditional Probability of an Event ⋆

We begin by considering the simple problem of computing the unconditional probability of an event $\boldsymbol{Z} = \boldsymbol{z}$. Although we can clearly use forward sampling for estimating this probability, we can also use unnormalized importance sampling, where the target distribution P is simply our prior distribution $P_{\mathcal{B}}(\mathcal{X})$, and the proposal distribution Q is the one defined by the mutilated network $\mathcal{B}_{\boldsymbol{Z}=\boldsymbol{z}}$. Our goal is to estimate the expectation of a function f, which is the indicator function of the query $\boldsymbol{z}$: $f(\xi) = \mathbf{1}\{\xi\langle\boldsymbol{Z}\rangle = \boldsymbol{z}\}$.

The unnormalized importance-sampling estimator for this case is simply:

$$\begin{aligned}\hat{P}_{\mathcal{D}}(\boldsymbol{z}) &= \frac{1}{M}\sum_{m=1}^{M}\mathbf{1}\{\xi[m]\langle\boldsymbol{Z}\rangle = \boldsymbol{z}\}w(\xi[m]) \\ &= \frac{1}{M}\sum_{m=1}^{M} w[m], \end{aligned} \tag{12.16}$$

where the equality follows because, by definition of Q, our sampling process generates samples $\xi[m]$ only where $\boldsymbol{z}$ holds.

When trying to bound the relative error of an estimator, a key quantity is the variance of the estimator *relative to its mean*. In the Chernoff bound, when we are estimating the probability p of a very low-probability event, the variance of the estimator, which is $p(1-p)$, is very high relative to the mean p. Importance sampling removes some of the variance associated with this sampling process, and it can therefore achieve better performance in certain cases.

In this case, the samples are derived from our proposal distribution Q, and the value of the function whose expectation we are computing is simply the weight. Thus, we need to bound the variance of the function $w(\mathcal{X})$ under our distribution Q. Let us consider the sampling process in the algorithm. As we go through the variables in the network, we encounter the observed variables $Z_1, \ldots, Z_k$. At each point, we multiply our current weight w by some conditional probability number $P_{\mathcal{B}}(Z_i = z_i \mid \mathrm{Pa}_{Z_i})$.

One situation where we can bound the variance arises in a restricted class of networks, one where the entries in the CPD of the variables Z_i are bounded away from the extremes of 0 and 1. More precisely, we assume that there is some pair of numbers $\ell > 0$ and $u < 1$ such that: for each variable $Z \in \boldsymbol{Z}$, $z \in Val(Z)$, and $\boldsymbol{u} \in Val(\mathrm{Pa}_Z)$, we have that $P_{\mathcal{B}}(Z = z \mid \mathrm{Pa}_Z = \boldsymbol{u}) \in [\ell, u]$. Next, we assume that $|\boldsymbol{Z}| = k$ for some small k. This assumption is not a trivial one; while queries often involve only a small number of variables, we often have a fairly large number of observations that we wish to incorporate.

Under these assumptions, the weight w generated through the LW process is necessarily in the interval ℓ^k and u^k. We can now redefine our weights by dividing each $w[m]$ by u^k:

$$w'[m] = w[m]/u^k.$$

Each weight $w'[m]$ is now a real-valued random variable in the range $[(\ell/u)^k, 1]$. For a data set $\mathcal{D}$ of weights $w[1], \ldots, w[M]$, we can now define:

$$\hat{p}'_{\mathcal{D}} = \frac{1}{M} \sum_{m=1}^{M} w'[m].$$

The key point is that the mean of this random variable, which is $P_{\mathcal{B}}(\boldsymbol{z})/u^k$, is therefore also in the range $[(\ell/u)^k, 1]$, and its variance is, at worst, the variance of a Bernoulli random variable with the same mean. Thus, we now have a random variable whose variance is not that small relative to its mean.

A simple generalization of Chernoff's bound (theorem A.4) to the case of real-valued variables can now be used to show that:

$$\begin{aligned} P_{\mathcal{D}}(\hat{P}_{\mathcal{D}}(\boldsymbol{z}) \notin P_{\mathcal{B}}(\boldsymbol{z})(1 \pm \epsilon)) &= P_{\mathcal{D}}(\hat{p}'_{\mathcal{D}} \notin \frac{1}{u^k} P_{\mathcal{B}}(\boldsymbol{z})(1 \pm \epsilon)) \\ &\leq 2e^{-M \frac{1}{u^k} P_{\mathcal{B}}(\boldsymbol{z}) \epsilon^2 / 3}. \end{aligned}$$

sample size

We can use this equation, as in the case of Bernoulli random variables, to derive a sufficient condition for the *sample size* that can guarantee that the estimator $\hat{P}_{\mathcal{D}}(\boldsymbol{z})$ of equation (12.16) has error at most ϵ with probability at least $1 - \delta$:

$$M \geq \frac{3 \ln(2/\delta) u^k}{P_{\mathcal{B}}(\boldsymbol{z}) \epsilon^2}. \tag{12.17}$$

Since $\mathcal{P}_{\mathcal{B}}(\boldsymbol{z}) \geq \ell^k$, a (stronger) sufficient condition is that:

$$M \geq \frac{3 \ln(2/\delta)}{\epsilon^2} \left(\frac{u}{\ell}\right)^k. \tag{12.18}$$

Chernoff bound

It is instructive to compare this bound to the one we obtain from the *Chernoff bound* in equation (12.5). The bound in equation (12.18) makes a weaker assumption about the probability of the event $\boldsymbol{z}$. Equation (12.5) requires that $P_{\mathcal{B}}(\boldsymbol{z})$ not be too low. By contrast, equation (12.17) assumes only that this probability is in a bounded range ℓ^k, u^k; the actual probability of the event $\boldsymbol{z}$ can still be very low — we have no guarantee on the actual magnitude of ℓ. Thus, for example, if our event $\boldsymbol{z}$ corresponds to a rare medical condition — one that has low probability given any instantiation of its parents — the estimator of equation (12.16) would give us a relative error bound, whereas standard sampling would not.

We can use this bound to determine in advance the number of samples required for a certain desired accuracy. A disadvantage of this approach is that it does not take into consideration the specific samples we happened to generate during our sampling process. Intuitively, not all samples contribute equally to the quality of the estimate. A sample whose weight is high is more compatible with the evidence $\boldsymbol{e}$, and it arguably provides us with more information. Conversely, a low-weight sample is not as informative, and a data set that contains a large number of low-weight samples might not be representative and might lead to a poor estimate. A somewhat more sophisticated approach is to preselect not the number of particles, but a predefined total weight. We then stop sampling when the total weight of the generated particles reaches our predefined lower bound.

Algorithm 12.3 Likelihood weighting with a data-dependent stopping rule

```
Procedure Data-Dependent-LW (
    B,    // Bayesian network over X
    Z = z,    // Instantiation of interest
    u,    // Upper bound on CPD entries of Z
    ε,    // Desired error bound
    δ     // Desired probability of error
)
1   γ ← 4(1+ε)/ε^2 ln 2/δ
2   k ← |Z|
3   W ← 0
4   M ← 0
5   while W < γ u^k
6     ξ, w ← LW-Sample(B, Z = z)
7     W ← W + w
8     M ← M + 1
9   return W/M
```

data-dependent likelihood weighting

For this algorithm, we can provide a similar theoretical analysis with certain guarantees for this *data-dependent likelihood weighting* approach. Algorithm 12.3 shows an algorithm that uses a data-dependent stopping rule to terminate the sampling process when enough weight has been accumulated. We can show that:

Theorem 12.1

Data-Dependent-LW *returns an estimate* $\hat{p}$ *for* $P_{\mathcal{B}}(\boldsymbol{Z} = \boldsymbol{z})$ *which, with probability at least* $1 - \delta$, *has a relative error of* ϵ.

expected sample size

We can also place an upper bound on the *expected sample size* used by the algorithm:

Theorem 12.2

The expected number of samples used by Data-Dependent-LW *is*

$$\frac{u^k}{P_{\mathcal{B}}(\boldsymbol{z})}\gamma \leq \left(\frac{u}{\ell}\right)^k \gamma,$$

where $\gamma = \frac{4(1+\epsilon)}{\epsilon^2} \ln \frac{2}{\delta}$.

The intuition behind this result is straightforward. The algorithm terminates when $W \geq \gamma u^k$. The expected contribution of each sample is $\boldsymbol{E}_{Q(\mathcal{X})}[w(\xi)] = P_{\mathcal{B}}(\boldsymbol{z})$. Thus, the total number of samples required to achieve a total weight of $W \geq \gamma u^k$ is $M \geq \gamma u^k / P_{\mathcal{B}}(\boldsymbol{z})$. Although this bound on the expected number of samples is no better than our bound in equation (12.17), the data-dependent bound allows us to stop early in cases where we were lucky in our random choice of samples, and to continue sampling in cases where we were unlucky.

12.2.3.3 Ratio Likelihood Weighting

ratio likelihood weighting

We now move to the problem of computing a conditional probability $P(\boldsymbol{y} \mid \boldsymbol{e})$ for a specific event $\boldsymbol{y}$. One obvious approach is *ratio likelihood weighting*: we compute the conditional

probability as $P(\boldsymbol{y}, \boldsymbol{e})/P(\boldsymbol{e})$, and use unnormalized importance sampling (equation (12.16)) for both the numerator and denominator.

We can therefore estimate the conditional probability $P(\boldsymbol{y} \mid \boldsymbol{e})$ in two phases: We use the algorithm of algorithm 12.2 M times with the argument $\boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{E} = \boldsymbol{e}$, to generate one set $\mathcal{D}$ of weighted samples $(\xi[1], w[1]), \ldots, (\xi[M], w[M])$. We use the same algorithm M' times with the argument $\boldsymbol{E} = \boldsymbol{e}$, to generate another set $\mathcal{D}'$ of weighted samples $(\xi'[1], w'[1]), \ldots, (\xi'[M'], w'[M'])$. We can then estimate:

$$\hat{P}_{\mathcal{D}}(\boldsymbol{y} \mid \boldsymbol{e}) = \frac{\hat{P}_{\mathcal{D}}(\boldsymbol{y}, \boldsymbol{e})}{\hat{P}_{\mathcal{D}'}(\boldsymbol{e})} = \frac{1/M \sum_{m=1}^{M} w[m]}{1/M' \sum_{m=1}^{M'} w'[m]}. \tag{12.19}$$

In ratio LW, the numerator and denominator are both using unnormalized importance sampling, which admits a rigorous theoretical analysis. Thus, we can now provide bounds on the number of samples M required to obtain a good estimate for both $P(\boldsymbol{y}, \boldsymbol{e})$ and $P(\boldsymbol{e})$.

12.2.3.4 Normalized Likelihood Weighting

Ratio LW allows us to estimate the probability of a single query $P(\boldsymbol{y} \mid \boldsymbol{e})$. In many cases, however, we are interested in estimating an entire joint distribution $P(\boldsymbol{Y} \mid \boldsymbol{e})$ for some variable or subset of variables $\boldsymbol{Y}$. We can answer such a query by running ratio LW for each $\boldsymbol{y} \in Val(\boldsymbol{Y})$, but this approach is typically too computationally expensive to be practical.

normalized likelihood weighting

An alternative approach is to use *normalized likelihood weighting*, which is based on the normalized importance sampling estimator of equation (12.13). In this application, our target distribution is $P(\mathcal{X}) = P_{\mathcal{B}}(\mathcal{X} \mid \boldsymbol{e})$. As we mentioned, we do not have access to P directly; rather, we can evaluate $\tilde{P}(\mathcal{X}) = P_{\mathcal{B}}(\mathcal{X}, \boldsymbol{e})$, which is the probability of a full assignment and can be easily computed via the chain rule. In this case, we are trying to estimate the expectation of a function f which is the indicator function of the query $\boldsymbol{y}$: $f(\xi) = \boldsymbol{I}\{\xi\langle \boldsymbol{Y} \rangle = \boldsymbol{y}\}$. Applying the normalized importance sampling estimator of equation (12.13) to this setting, we obtain precisely the estimator of equation (12.6).

The quality of the importance sampling estimator depends largely on how close the proposal distribution Q is to the target distribution P. We can gain intuition for this question by considering two extreme cases. If all of the evidence in our network is at the roots, the proposal distribution is precisely the posterior, and there is no need to compensate; indeed, no evidence is encountered along the way, and all samples will have the same weight $P(\boldsymbol{e})$. On the other side of the spectrum, if all of the evidence is at the leaves, our proposal distribution $Q(\mathcal{X})$ is the prior distribution $P_{\mathcal{B}}(\mathcal{X})$, leaving the correction purely to the weights. In this situation, LW will work reasonably only if the prior is similar to the posterior. Otherwise, most of our samples will be irrelevant, a fact that will be reflected by their low weight. For example, consider a medical-diagnosis setting, and assume that our evidence is a very unusual combination of symptoms generated by only one very rare disease. Most samples will not involve this disease and will give only very low probability to this combination of symptoms. Indeed, the combinations sampled are likely to be irrelevant and are not useful at all for understanding what disease the patient has. We return to this issue in section 12.2.4.

To understand the relationship between the prior and the posterior, note that the prior is a

weighted average of the posteriors, weighted over different instantiations of the evidence:

$$P(\mathcal{X}) = \sum_{\boldsymbol{e}} P(\boldsymbol{e}) P(\mathcal{X} \mid \boldsymbol{e}).$$

If the evidence is very likely, then it is a major component in this summation, and it is probably not too far from the prior. For example, in the network $\mathcal{B}^{student}$, the event $S = s^1$ is fairly likely, and the posterior distribution $P_{\mathcal{B}^{student}}(\mathcal{X} \mid s^1)$ is fairly similar to the prior. However, for unlikely evidence, the weight of $P(\mathcal{X} \mid \boldsymbol{e})$ is negligible, and there is nothing constraining the posterior to be similar to the prior. Indeed, our distribution $P_{\mathcal{B}^{student}}(\mathcal{X} \mid l^0)$ is very different from the prior.

Unfortunately, there is currently no formal analysis for the number of particles required to achieve a certain quality of estimate using normalized importance sampling. In many cases, we simply preselect a number of particles that seems large enough, and we generate that number. Alternatively, we can use a heuristic approach that uses the total weight of the particles generated so far as guidance as to the extent to which they are representative. Thus, for example, we might decide to generate samples until a certain minimum bound on the total weight has been reached, as in Data-Dependent-LW. We note, however, that this approach is entirely heuristic in this case (as in all cases where we do not have bounds $[\ell, u]$ on our CPDs). Furthermore, there are cases where the evidence is simply unlikely in all configurations, and therefore all samples will have low weights.

12.2.3.5 Conditional Probabilities: Comparison

We have seen two variants of likelihood weighting: normalized LW and ratio LW. Ratio LW has two related advantages. The normalized LW process samples an assignment of the variables $\boldsymbol{Y}$ (those not in $\boldsymbol{E}$), whereas ratio LW simply sets the values of these variables. The additional sampling step for $\boldsymbol{Y}$ introduces additional variance into the overall process, leading to a reduction in the robustness of the estimate. Thus, in many cases, the variance of this estimator is lower than that of equation (12.6), leading to more robust estimates.

A second advantage of ratio LW is that it is much easier to analyze, and therefore it is associated with stronger guarantees regarding the number of samples required to get a good estimate. However, these bounds are useful only under very strong conditions: a small number of evidence variables, and a bound on the skew of the CPD entries in the network.

On the other hand, a significant disadvantage of ratio LW is the fact that each query $\boldsymbol{y}$ requires that we generate a new set of samples for the event $\boldsymbol{y}, \boldsymbol{e}$. It is often the case that we want to evaluate the probability of multiple queries relative to the same set of evidence. The normalized LW approach allows these multiple computations to be executed relative to the same set of samples, whereas ratio LW requires a separate sample set for each query $\boldsymbol{y}$. This cost is particularly problematic when we are interested in computing the joint distribution over a subset of variables. Probably due to this last point, normalized LW is used more often in practice.

12.2.4 Importance Sampling Revisited

The likelihood weighting algorithm uses, as its proposal distribution, the very simple distribution obtained from mutilating the network by eliminating edges incoming to observed variables. However, this proposal distribution can be far from optimal. For example, if the CPDs associated

with these evidence variables are skewed, the importance weights are likely to be quite large, resulting in estimators with high variance. Indeed, somewhat surprisingly, even in very simple cases, the obvious proposal distribution may not be optimal. For example, if X is not a root node in the network, the optimal proposal distribution for computing $P(X = x)$ may not be the distribution P, even without evidence! (See exercise 12.5.)

backward importance sampling

The importance sampling framework is very general, however, and several other proposal distributions have been utilized. For example, *backward importance sampling* generates samples for parents of evidence variables using the likelihood of their children. Most simply, if X is a variable whose child Y is observed to be $Y = y$, we might generate some samples for X from a renormalized distribution $Q(X) \propto P(Y = y \mid X)$. We can continue this process, sampling X's parents from the likelihood of X's sampled value. We can also propose more complex schemes that sample the value of a variable given a combination of sampled or observed values for some of its parents and/or children. One can also consider hybrid approaches that use some global approximate inference algorithm (such as those in chapter 11) to construct a proposal distribution, which is then used as the basis for sampling. **As long as the importance weights are computed correctly, we are guaranteed that this process is correct.** (See exercise 12.7.) This process can lead to significant improvements in theory, and it does lead to improvements in some cases in practice.

12.3 Markov Chain Monte Carlo Methods

One of the limitations of likelihood weighting is that an evidence node affects the sampling only for nodes that are its descendants. The effect on nodes that are nondescendants is accounted for only by the weights. As we discussed, in cases where much of the evidence is at the leaves of the network, we are essentially sampling from the prior distribution, which is often very far from the desired posterior. We now present an alternative sampling approach that generates a *sequence* of samples. This sequence is constructed so that, although the first sample may be generated from the prior, successive samples are generated from distributions that provably get closer and closer to the desired posterior. We note that, unlike forward sampling methods (including likelihood weighting), Markov chain methods apply equally well to directed and to undirected models. Indeed, the algorithm is easier to present in the context of a distribution P_Φ defined in terms of a general set of factors Φ.

12.3.1 Gibbs Sampling Algorithm

Gibbs sampling

One idea for addressing the problem with forward sampling approaches is to try to "fix" the sample we generated by resampling some of the variables we generated early in the process. Perhaps the simplest method for doing this is presented in algorithm 12.4. This method, called *Gibbs sampling*, starts out by generating a sample of the unobserved variables from some initial distribution; for example, we may use the mutilated network to generate a sample using forward sampling. Starting from that sample, we then iterate over each of the unobserved variables, sampling a new value for each variable *given* our current sample for all other variables. This process allows information to "flow" across the network as we sample each variable.

To apply this algorithm to a network with evidence, we first reduce all of the factors by the observations $\boldsymbol{e}$, so that the distribution P_Φ used in the algorithm corresponds to $P(\boldsymbol{X} \mid \boldsymbol{e})$.

Algorithm 12.4 Generating a Gibbs chain trajectory

```
Procedure Gibbs-Sample (
    X      // Set of variables to be sampled
    Φ      // Set of factors defining P_Φ
    P^(0)(X),    // Initial state distribution
    T      // Number of time steps
)
1   Sample x^(0) from P^(0)(X)
2   for t = 1, ..., T
3     x^(t) ← x^(t−1)
4     for each X_i ∈ X
5       Sample x_i^(t) from P_Φ(X_i | x_{−i})
6         // Change X_i in x^(t)
7   return x^(0), ..., x^(T)
```

Example 12.4

Let us revisit example 12.3, recalling that we have the observations s^1, l^0. In this case, our algorithm will generate samples over the variables D, I, G. The set of reduced factors Φ is therefore: $P(I), P(D), P(G \mid I, D), P(s^1 \mid I), P(l^0 \mid G)$. Our algorithm begins by generating one sample, say by forward sampling. Assume that this sample is $d^{(0)} = d^1, i^{(0)} = i^0, g^{(0)} = g^2$. In the first iteration, it would now resample all of the unobserved variables, one at a time, in some predetermined order, say G, I, D. Thus, we first sample $g^{(1)}$ from the distribution $P_\Phi(G \mid d^1, i^0)$.

Note that because we are computing the distribution over a single variable given all the others, this computation can be performed very efficiently:

$$\begin{aligned} P_\Phi(G \mid d^1, i^0) &= \frac{P(i^0)P(d^1)P(G \mid i^0, d^1)P(l^0 \mid G)P(s^1 \mid i^0)}{\sum_g P(i^0)P(d^1)P(g \mid i^0, d^1)P(l^0 \mid g)P(s^1 \mid i^0)} \\ &= \frac{P(G \mid i^0, d^1)P(l^0 \mid G)}{\sum_g P(g \mid i^0, d^1)P(l^0 \mid g)}. \end{aligned}$$

Thus, we can compute the distribution simply by multiplying all factors that contain G, with all other variables instantiated, and renormalizing to obtain a distribution over G.

Having sampled $g^{(1)} = g^3$, we now continue to resampling $i^{(1)}$ from the distribution $P_\Phi(I \mid d^1, g^3)$, obtaining, for example, $i^{(1)} = i^1$; note that the distribution for I is conditioned on the newly sampled value $g^{(1)}$. Finally, we sample $d^{(1)}$ from $P_\Phi(D \mid g^3, i^1)$, obtaining d^1. The result of the first iteration of sampling is, then, the sample (i^1, d^1, g^3). The process now repeats. ■

Note that, unlike forward sampling, the sampling process for G takes into consideration the downstream evidence at its child L. Thus, its sampling distribution is arguably closer to the posterior. Of course, it is not the true posterior, since it still conditions on the originally sampled values for I, D, which were sampled from the prior distribution. However, we now resample I and D from a distribution that conditions on the new value of G, so one can imagine that their sampling distribution may also be closer to the posterior. Thus, perhaps the next sample of G,

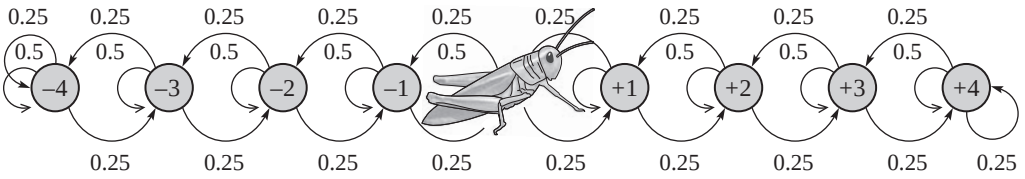


Figure 12.3 The Grasshopper **Markov chain**

which uses these new values for I, D (and conditions on the evidence l^0), will be sampled from a distribution even closer to the posterior. Indeed, this intuition is correct. One can show that, as we repeat this sampling process, the distribution from which we generate each sample gets closer and closer to the posterior $P_\Phi(\boldsymbol{X}) = P(\boldsymbol{X} \mid \boldsymbol{e})$.

Markov chain
Monte Carlo

In the subsequent sections, we formalize this intuitive argument using a framework called *Markov chain Monte Carlo (MCMC)*. This framework provides a general approach for generating samples from the posterior distribution, in cases where we cannot efficiently sample from the posterior directly. In MCMC, we construct an iterative process that gradually samples from distributions that are closer and closer to the posterior. A key question is, of course, how many iterations we should perform before we can collect a sample as being (almost) generated from the posterior. In the following discussion, we provide the formal foundations for MCMC algorithms, and we try to address this and other important questions. We also present several valuable generalizations.

12.3.2 Markov Chains

12.3.2.1 Basic Definition

At a high level, a Markov chain is defined in terms of a graph of states over which the sampling algorithm takes a random walk. In the case of graphical models, this graph is *not* the original graph, but rather a graph whose nodes are the possible assignments to our variables $\boldsymbol{X}$.

Definition 12.2

Markov chain

transition model

A Markov chain *is defined via a state space* $Val(\boldsymbol{X})$ *and a model that defines, for every state* $\boldsymbol{x} \in Val(X)$ *a* next-state *distribution over* $Val(X)$. *More precisely, the* transition model $\mathcal{T}$ *specifies for each pair of state* $\boldsymbol{x}, \boldsymbol{x}'$ *the probability* $\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}')$ *of going from* $\boldsymbol{x}$ *to* $\boldsymbol{x}'$. *This transition probability applies whenever the chain is in state* $\boldsymbol{x}$. ■

homogeneous Markov chain

We note that, in this definition and in the subsequent discussion, we restrict attention to *homogeneous*, where the system dynamics do not change over time.

We illustrate this concept with a simple example.

Example 12.5

Consider a Markov chain whose states consist of the nine integers $-4, \ldots, +4$, *arranged as points on a line. Assume that a drunken grasshopper starts out in position* 0 *on the line. At each point in time, it stays where it is with probability* 0.5, *or it jumps left and right with equal probability. Thus,* $\mathcal{T}(i \to i) = 0.5$, $\mathcal{T}(i \to i+1) = 0.25$, *and* $\mathcal{T}(i \to i-1) = 0.25$. *However, the two end positions are blocked by walls; hence, if the grasshopper is in position* $+4$ *and tries to jump right, it*

remains in position $+4$. Thus, for example, $\mathcal{T}(+4 \rightarrow +4) = 0.75$. We can visualize the state space as a graph, with probability-weighted directed edges corresponding to transitions between different states. The graph for our example is shown in figure 12.3. ■

We can imagine a random sampling process, that defines a random sequence of states $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots$. Because the transition model is random, the state of the process at step t can be viewed as a random variable $\boldsymbol{X}^{(t)}$. We assume that the initial state $\boldsymbol{X}^{(0)}$ is distributed according to some initial state distribution $P^{(0)}(\boldsymbol{X}^{(0)})$. We can now define distributions over the subsequent states $P^{(1)}(\boldsymbol{X}^{(1)}), P^{(2)}(\boldsymbol{X}^{(2)}), \ldots$ using the chain dynamics:

$$P^{(t+1)}(\boldsymbol{X}^{(t+1)} = \boldsymbol{x}') = \sum_{\boldsymbol{x} \in Val(\boldsymbol{X})} P^{(t)}(\boldsymbol{X}^{(t)} = \boldsymbol{x}) \mathcal{T}(\boldsymbol{x} \rightarrow \boldsymbol{x}'). \tag{12.20}$$

Intuitively, the probability of being at state $\boldsymbol{x}'$ at time $t+1$ is the sum over all possible states $\boldsymbol{x}$ that the chain could have been in at time t of the probability being in state $\boldsymbol{x}$ times the probability that the chain took a transition from $\boldsymbol{x}$ to $\boldsymbol{x}'$.

12.3.2.2 Asymptotic Behavior

For our purposes, the most important aspect of a Markov chain is its long-term behavior.

Example 12.6

Because the grasshopper's motion is random, we can consider its location at time t to be a random variable, which we denote $X^{(t)}$. Consider the distribution over $X^{(t)}$. Initially, the grasshopper is at 0, so that $P(X^{(0)} = 0) = 1$. At time 1, we have that $X^{(1)}$ is 0 with probability 0.5, and $+1$ or -1, each with probability 0.25. At time 2, we have that $X^{(2)}$ is 0 with probability $0.5^2 + 2 \cdot 0.25^2 = 0.375$, $+1$ and -1 each with probability $2(0.5 \cdot 0.25) = 0.25$, and $+2$ and -2 each with probability $0.25^2 = 0.0625$. As the process continues, the probability gets spread out over more and more of the states. For example, at time $t = 10$, the probabilities of the different states range from 0.1762 for the value 0, and 0.0518 for the values ± 4. At $t = 50$, the distribution is almost uniform, with a range of 0.1107–0.1116. ■

Thus, one approach for sampling from the uniform distribution over the set $-4, \ldots, +4$ is to start off at 0 and then randomly choose the next state from the transition model for this chain. After some number of such steps t, our state $X^{(t)}$ would be sampled from a distribution that is very close to uniform over this space. We note that this approach is not a very good one for sampling from a uniform distribution; indeed, the expected time required for such a chain even to reach the boundaries of the interval $[-K, K]$ is K^2 steps. However, this general approach applies much more broadly, including in cases where our "long-term" distribution is not one from which we can easily sample.

MCMC sampling

Markov chain Monte carlo (MCMC) sampling is a process that mirrors the dynamics of the Markov chain; the process of generating an MCMC trajectory is shown in algorithm 12.5. The sample $\boldsymbol{x}^{(t)}$ is drawn from the distribution $P^{(t)}$. We are interested in the limit of this process, that is, whether $P^{(t)}$ converges, and if so, to what limit.

Algorithm 12.5 Generating a Markov chain trajectory

```
Procedure MCMC-Sample (
  P^(0)(X),    // Initial state distribution
  T,           // Markov chain transition model
  T            // Number of time steps
)
1   Sample x^(0) from P^(0)(X)
2   for t = 1, ..., T
3     Sample x^(t) from T(x^(t-1) → X)
4     return x^(0), ..., x^(T)
```

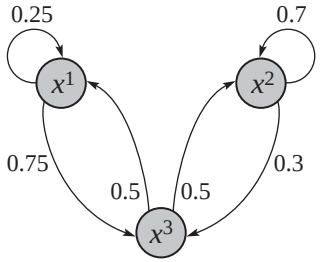


Figure 12.4 A simple Markov chain

12.3.2.3 Stationary Distributions

Intuitively, as the process converges, we would expect $P^{(t+1)}$ to be close to $P^{(t)}$. Using equation (12.20), we obtain:

$$P^{(t)}(\boldsymbol{x}') \approx P^{(t+1)}(\boldsymbol{x}') = \sum_{\boldsymbol{x} \in Val(\boldsymbol{X})} P^{(t)}(\boldsymbol{x}) \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}').$$

At convergence, we would expect the resulting distribution $\pi(\boldsymbol{X})$ to be an equilibrium relative to the transition model; that is, the probability of being in a state is the same as the probability of transitioning into it from a randomly sampled predecessor. Formally:

Definition 12.3 stationary distribution

A distribution $\pi(\boldsymbol{X})$ is a stationary distribution *for a Markov chain $\mathcal{T}$ if it satisfies:*

$$\pi(\boldsymbol{X} = \boldsymbol{x}') = \sum_{\boldsymbol{x} \in Val(\boldsymbol{X})} \pi(\boldsymbol{X} = \boldsymbol{x}) \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}'). \tag{12.21}$$

■

A stationary distribution is also called an *invariant distribution*.[2]

2. If we view the transition model as a matrix defined as $A_{i,j} = \mathcal{T}(\boldsymbol{x}_i \to \boldsymbol{x}_j)$, then a stationary distribution is an eigen-vector of the matrix, corresponding to the eigen-value 1. In general, many aspects of the theory of Markov chains have an algebraic interpretation in terms of matrices and vectors.

As we have already discussed, the uniform distribution is a stationary distribution for the Markov chain of example 12.5. To take a slightly different example:

Example 12.7

Figure 12.4 shows an example of a different simple Markov chain where the transition probabilities are less uniform. By definition, the stationary distribution π must satisfy the following three equations:

$$\begin{aligned}\pi(x^1) &= 0.25\pi(x^1) + 0.5\pi(x^3)\\ \pi(x^2) &= 0.7\pi(x^2) + 0.5\pi(x^3)\\ \pi(x^3) &= 0.75\pi(x^1) + 0.3\pi(x^2),\end{aligned}$$

as well as the one asserting that it is a legal distribution:

$$\pi(x^1) + \pi(x^2) + \pi(x^3) = 1.$$

It is straightforward to verify that this system has a unique solution: $\pi(x^1) = 0.2$, $\pi(x^2) = 0.5$, $\pi(x^3) = 0.3$. For example, the first equation asserts that

$$0.2 = 0.25 \cdot 0.2 + 0.5 \cdot 0.3,$$

which clearly holds. ■

In general, there is no guarantee that our MCMC sampling process converges to a stationary distribution.

Example 12.8

Consider the Markov chain over two states x^1 and x^2, such that $\mathcal{T}(x^1 \to x^2) = 1$ and $\mathcal{T}(x^2 \to x^1) = 1$. If $P^{(0)}$ is such that $P^{(0)}(x^1) = 1$, then the step t distribution $P^{(t)}$ has $P^{(t)}(x^1) = 1$ if t is even, and $P^{(t)}(x^2) = 1$ if t is odd. Thus, there is no convergence to a stationary distribution. ■

periodic Markov chain

Markov chains such as this, which exhibit a fixed cyclic behavior, are called *periodic Markov chains*.

There is also no guarantee that the stationary distribution is unique: In some chains, the stationary distribution reached depends on our starting distribution $P^{(0)}$. Situations like this occur when the chain has several distinct regions that are not reachable from each other. Chains such as this are called *reducible Markov chains*.

reducible Markov chain

We wish to restrict attention to Markov chains that have a unique stationary distribution, which is reached from any starting distribution $P^{(0)}$. There are various conditions that suffice to guarantee this property. The condition most commonly used is a fairly technical one: that the chain be *ergodic*. In the context of Markov chains where the state space $Val(\boldsymbol{X})$ is finite, the following condition is equivalent to this requirement:

ergodic Markov chain

Definition 12.4

A Markov chain is said to be regular *if there exists some number k such that, for every $\boldsymbol{x}, \boldsymbol{x}' \in Val(\boldsymbol{X})$, the probability of getting from $\boldsymbol{x}$ to $\boldsymbol{x}'$ in exactly k steps is > 0.* ■

regular Markov chain

In our Markov chain of example 12.5, the probability of getting from any state to any state in exactly 9 steps is greater than 0. Thus, this Markov chain is regular. Similarly, in the Markov chain of example 12.7, we can get from any state to any state in exactly two steps.

The following result can be shown to hold:

Theorem 12.3 *If a finite state Markov chain $\mathcal{T}$ is regular, then it has a unique stationary distribution.*

Ensuring regularity is usually straightforward. Two simple conditions that together guarantee regularity in finite-state Markov chains are as follows. First, it is possible to get from any state to any state using a positive probability path in the state graph. Second, for each state $\boldsymbol{x}$, there is a positive probability of transitioning from $\boldsymbol{x}$ to $\boldsymbol{x}$ in one step (a self-loop). These two conditions together are sufficient but not necessary to guarantee regularity (see exercise 12.12). However, they often hold in the chains used in practice.

12.3.2.4 Multiple Transition Models

In the case of graphical models, our state space has a factorized structure — each state is an assignment to several variables. When defining a transition model over this state space, we can consider a fully general case, where a transition can go from any state to any state. However, it is often convenient to decompose the transition model, considering transitions that update only a single component of the state vector at a time, that is, only a value for a single variable.

Example 12.9 *Consider an extension to our* Grasshopper *chain, where the grasshopper lives, not on a line, but in a two-dimensional plane. In this case, the state of the system is defined via a pair of random variables X, Y. Although we could define a joint transition model over both dimensions simultaneously, it might be easier to have separate transition models for the X and Y coordinate.* ■

kernel

In this case, as in several other settings, we often define a set of transition models, each with its own dynamics. Each such transition model $\mathcal{T}_i$ is called a *kernel*. In certain cases, the different kernels are necessary, because no single kernel on its own suffices to ensure regularity. This is the case in example 12.9. In other cases, having multiple kernels simply makes the state space more "connected" and therefore speeds the convergence to a stationary distribution.

multi-kernel Markov chain

There are several ways of constructing a single Markov chain from *multiple kernels*. One common approach is simply to select randomly between them at each step, using any distribution. Thus, for example, at each step, we might select one of $\mathcal{T}_1, \ldots, \mathcal{T}_k$, each with probability $1/k$. Alternatively, we can simply cycle over the different kernels, taking each one in turn. Clearly, this approach does not define a homogeneous chain, since the kernel used in step i is different from the one used in step $i+1$. However, we can simply view the process as defining a single chain $\mathcal{T}$, each of whose steps is an aggregate step, consisting of first taking $\mathcal{T}_1$, then $\mathcal{T}_2$, ..., through $\mathcal{T}_k$.

In the case of graphical models, one approach is to define a multikernel chain, where we have a kernel $\mathcal{T}_i$ for each variable $X_i \in \boldsymbol{X}$. Let $\boldsymbol{X}_{-i} = \mathcal{X} - \{X_i\}$, and let $\boldsymbol{x}_i$ denote an instantiation to $\boldsymbol{X}_i$. The model $\mathcal{T}_i$ takes a state $(\boldsymbol{x}_{-i}, x_i)$ and transitions to a state of the form $(\boldsymbol{x}_{-i}, x_i')$. As we discussed, we can combine the different kernels into a single global model in various ways.

Regardless of the structure of the different kernels, **we can prove that a distribution is a stationary distribution for the multiple kernel chain by proving that it is a stationary distribution (satisfies equation (12.21)) for each of individual kernels $\mathcal{T}_i$. Note that each kernel by itself is generally not ergodic; but as long as each kernel satisfies certain conditions (specified in definition 12.5) that imply that it has the desired stationary distribution, we can combine them to produce a coherent chain, which may be ergodic as a whole. This**

ability to add new types of transitions to our chain is an important asset in dealing with the issue of local maxima, as we will discuss.

12.3.3 Gibbs Sampling Revisited

The theory of Markov chains provides a general framework for generating samples from a target distribution π. In this section, we discuss the application of this framework to the sampling tasks encountered in probabilistic graphical models. In this case, we typically wish to generate samples from the posterior distribution $P(\boldsymbol{X} \mid \boldsymbol{E} = \boldsymbol{e})$, where $\boldsymbol{X} = \mathcal{X} - \boldsymbol{E}$. Thus, we wish to define a chain for which $P(\boldsymbol{X} \mid \boldsymbol{e})$ is the stationary distribution. Thus, we define the states of the Markov chain to be instantiations $\boldsymbol{x}$ to $\mathcal{X} - \boldsymbol{E}$. In order to define a Markov chain, we need to define a process that transitions from one state to the other, converging to a stationary distribution $\pi(\boldsymbol{X})$, which is the desired posterior distribution $P(\boldsymbol{X} \mid \boldsymbol{e})$. As in our earlier example, we assume that $P(\boldsymbol{X} \mid \boldsymbol{e}) = P_\Phi$ for some set of factors Φ that are defined by reducing the original factors in our graphical model by the evidence $\boldsymbol{e}$. This reduction allows us to simplify notation and to discuss the methods in a way that applies both to directed and undirected graphical models.

Gibbs chain

Gibbs sampling is based on one yet effective *Markov chain* for factored state spaces, which is particularly efficient for graphical models. We define the kernel $\mathcal{T}_i$ as follows. Intuitively, we simply "forget" the value of X_i in the current state and sample a new value for X_i from its posterior given the rest of the current state. More precisely, let $(\boldsymbol{x}_{-i}, x_i)$ be a state in the chain. We define:

$$\mathcal{T}_i((\boldsymbol{x}_{-i}, x_i) \to (\boldsymbol{x}_{-i}, x_i')) = P(x_i' \mid \boldsymbol{x}_{-i}). \tag{12.22}$$

Note that the transition probability does not depend on the current value x_i of X_i, but only on the remaining state $\boldsymbol{x}_{-i}$. It is not difficult to show that the posterior distribution $P_\Phi(\boldsymbol{X}) = P(\mathcal{X} \mid \boldsymbol{e})$ is a *stationary distribution* of this process. (See exercise 12.13.)

Gibbs stationary distribution

The sampling algorithm for a single trajectory of the Gibbs chain was shown earlier in this section, in algorithm 12.4. Recall that the Gibbs chain is defined via a set of kernels; we use the multistep approach to combine them. Thus, the different local kernels are taken consecutively; having changed the value for a variable X_1, the value for X_2 is sampled based on the new value. Note that a step in the aggregate chain occurs only once we have executed every local transition once.

Gibbs sampling is particularly easy to implement in the many graphical models where we can compute the transition probability $P(X_i \mid \boldsymbol{x}_{-i})$ (in line 5 of the algorithm) very efficiently.

Markov blanket

In particular, as we now show, this distribution can be done based only on the *Markov blanket* of X_i. We show this analysis for a Markov network; the application to Bayesian networks is straightforward. Recalling definition 4.4, we have that:

$$\begin{aligned} P_\Phi(\boldsymbol{X}) &= \frac{1}{Z} \prod_j \phi_j(\boldsymbol{D}_j) \\ &= \frac{1}{Z} \prod_{j \,:\, X_i \in \boldsymbol{D}_j} \phi_j(\boldsymbol{D}_j) \prod_{j \,:\, X_i \notin \boldsymbol{D}_j} \phi_j(\boldsymbol{D}_j). \end{aligned}$$

Let $\boldsymbol{x}_{j,-i}$ denote the assignment in $\boldsymbol{x}_{-i}$ to $\boldsymbol{D}_j - \{X_i\}$, noting that when $X_i \notin \boldsymbol{D}_j$, $\boldsymbol{x}_{j,-i}$ is a

full assignment to $\boldsymbol{D}_j$. We can now derive:

$$\begin{aligned} P(x_i' \mid \boldsymbol{x}_{-i}) &= \frac{P(x_i', \boldsymbol{x}_{-i})}{\sum_{x_i''} P(x_i'', \boldsymbol{x}_{-i})} \\ &= \frac{\frac{1}{Z}\prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i', \boldsymbol{x}_{j,-i}) \prod_{\boldsymbol{D}_j \not\ni X_i} \phi_j(x_i', \boldsymbol{x}_{j,-i})}{\frac{1}{Z}\sum_{x_i''} \prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i'', \boldsymbol{x}_{j,-i}) \prod_{\boldsymbol{D}_j \not\ni X_i} \phi_j(x_i'', \boldsymbol{x}_{j,-i})} \\ &= \frac{\prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i', \boldsymbol{x}_{j,-i}) \prod_{\boldsymbol{D}_j \not\ni X_i} \phi_j(\boldsymbol{x}_{j,-i})}{\sum_{x_i''} \prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i'', \boldsymbol{x}_{j,-i}) \prod_{\boldsymbol{D}_j \not\ni X_i} \phi_j(\boldsymbol{x}_{j,-i})} \\ &= \frac{\prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i', \boldsymbol{x}_{j,-i})}{\sum_{x_i''} \prod_{\boldsymbol{D}_j \ni X_i} \phi_j(x_i'', \boldsymbol{x}_{j,-i})}. \end{aligned} \tag{12.23}$$

This last expression uses only the factors involving X_i, and depends only on the instantiation in $\boldsymbol{x}_{-i}$ of X_i's Markov blanket. In the case of Bayesian networks, this expression reduces to a formula involving only the CPDs of X_i and its children, and its value, again, depends only on the assignment in $\boldsymbol{x}_{-i}$ to the Markov blanket of X_i.

Example 12.10

Consider again the Student *network of figure 12.1, with the evidence s^1, l^0. The kernel for the variable G is defined as follows. Given a state (i, d, g, s^1, l^0), we define $\mathcal{T}((i, g, d, s^1, l^0) \rightarrow (i, g', d, s^1, l^0)) = P(g' \mid i, d, s^1, l^0)$. This value can be computed locally, using only the CPDs that involve G, that is, the CPDs of G and L:*

$$P(g' \mid i, d, s^1, l^0) = \frac{P(g' \mid i, d) P(l^0 \mid g')}{\sum_{g''} P(g'' \mid i, d) P(l^0 \mid g'')}.$$

Similarly, the kernel for the variable I is defined to be $\mathcal{T}((i, g, d, s^1, l^0) \rightarrow (i', g, d, s^1, l^0)) = P(i' \mid g, d, s^1, l^0)$, which simplifies as follows:

$$P(i' \mid g, d, s^1, l^0) = \frac{P(i') P(g \mid i', d) P(s^1 \mid i')}{\sum_{i''} P(i'') P(g \mid i'', d) P(s^1 \mid i'')}.$$

■

As presented, the algorithm is defined via a sequence of local kernels, where each samples a single variable conditioned on all the rest. The reason for this approach is computational. As we showed, we can easily compute the transition model for a single variable given the rest. However, there are cases where we can simultaneously sample several variables efficiently. Specifically, assume we can partition the variables $\boldsymbol{X}$ into several disjoint *blocks* of variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$, such that we can efficiently sample $\boldsymbol{x}_i$ from $P_\Phi(\boldsymbol{X}_i \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_k)$. In this case, we can modify our Gibbs sampling algorithm to iteratively sample blocks of variables, rather than individual variables, thereby taking much "longer-range" transitions in the state space in a single sampling step. Here, like in Gibbs sampling, we define the algorithm to be producing a new sample only once all blocks have been resampled. This algorithm is called *block Gibbs*. Note that standard Gibbs sampling is a special case of block Gibbs sampling, with the blocks corresponding to individual variables.

block Gibbs sampling

Example 12.11

Consider the Bayesian network induced by the plate model of example 6.11. Here, we generally have n students, each with a variable representing his or her intelligence, and m courses, each

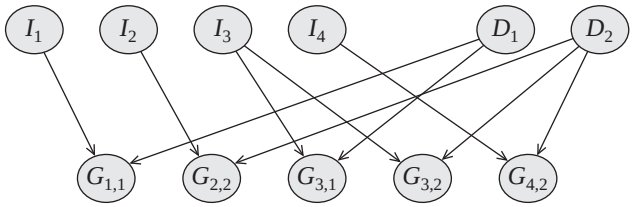


Figure 12.5 A Bayesian network with four students, two courses, and five grades

with a variable representing its difficulty. We also have a set of grades for students in classes (not necessarily a grade for each student in every class). Using an abbreviated notation, we have a set of variables $I_1, \ldots, I_n$ for the students (where each $I_j = I(s_j)$), $\boldsymbol{D} = \{D_1, \ldots, D_\ell\}$ for the courses, and $\boldsymbol{G} = \{G_{j,k}\}$ for the grades, where each variable $G_{j,k}$ has the parents I_j and D_k. See figure 12.5 for an example with $n = 4$ and $\ell = 2$. Let us assume that we observe the grades, so that we have evidence $\boldsymbol{G} = \boldsymbol{g}$. An examination of active paths shows that the different variables I_j are conditionally independent given an assignment $\boldsymbol{d}$ to $\boldsymbol{D}$. Thus, given $\boldsymbol{D} = \boldsymbol{d}, \boldsymbol{G} = \boldsymbol{g}$, we can efficiently sample all of the $\boldsymbol{I}$ variables as a block by sampling each I_j independently of the others. Similarly, we can sample all of the $\boldsymbol{D}$ variables as a block given an assignment $\boldsymbol{I} = \boldsymbol{i}, \boldsymbol{G} = \boldsymbol{g}$. Thus, we can alternate steps where in one we sample $\boldsymbol{i}[m]$ given $\boldsymbol{g}$ and $\boldsymbol{d}[m]$, and in the other we sample $\boldsymbol{d}[m+1]$ given $\boldsymbol{g}$ and $\boldsymbol{i}[m]$. ■

In this example, we can easily apply block Gibbs because the variables in each block are marginally independent given the variables outside the block. This independence property allows us to compute efficiently the conditional distribution $P_\Phi(\boldsymbol{X}_i \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_k)$, and to sample from it. Importantly, however, full independence is not essential: we need only have the property that the block-conditional distribution can be efficiently manipulated. For example, in a grid-structured network, we can easily define our blocks to consist of separate rows or of separate columns. In this case, the structure of each block is a simple chain-structured network; we can easily compute the conditional distribution of one row given all the others, and sample from it (see exercise 12.3).

We note that the Gibbs chain is not necessarily regular, and might not converge to a unique stationary distribution.

Example 12.12

Consider a simple network that consists of a single v-structure $X \rightarrow Z \leftarrow Y$, where the variables are all binary, X and Y are both uniformly distributed, and Z is the deterministic exclusive or of X and Y (that is, $Z = z^1$ iff $X \neq Y$). Consider applying Gibbs sampling to this network with the evidence z^1. The true posterior assigns probability $1/2$ to each of the two states x^1, y^0, z^1 and x^0, y^1, z^1. Assume that we start in the first of these two states. In this case, $P(X \mid y^0, z^1)$ assigns probability 1 to x^1, so that the X transition leaves the value of X unchanged. Similarly, the Y transition leaves the value of Y unchanged. Therefore, the chain will simply stay at the initial state forever, and it will never sample from the other state. The analogous phenomenon occurs for the other starting state. This chain is an example of a reducible Markov chain. ■

However, this chain is guaranteed to be regular whenever the distribution is positive, so that every value of X_i has positive probability given an assignment $\boldsymbol{x}_{-i}$ to the remaining variables.

Theorem 12.4 *Let $\mathcal{H}$ be a Markov network such that all of the clique potentials are strictly positive. Then the Gibbs-sampling Markov chain is regular.*

The proof is not difficult, and is left as an exercise (exercise 12.20).

Positivity is, however, not necessary; there are many examples of nonpositive distributions where the Gibbs chain is regular.

mixing

Importantly, however, even chains that are regular may require a long time to *mix*, that is, get close to the stationary distribution. In this case, instances generated from early in the sampling process will not be representative of the desired stationary distribution.

12.3.4 A Broader Class of Markov Chains ⋆

As we discussed, the use of MCMC methods relies on the construction of a Markov chain that has the desired properties: regularity, and the target stationary distribution. In the previous section, we described the Gibbs chain, a simple Markov chain that is guaranteed to have these properties under certain assumptions. However, Gibbs sampling is applicable only in certain circumstances; in particular, we must be able to sample from the distribution $P(X_i \mid \boldsymbol{x}_{-i})$. Although this sampling step is easy for discrete graphical models, in continuous models, the conditional distribution may not be one that has a parametric form that allows sampling, so that Gibbs is not applicable.

Even more important, **the Gibbs chain uses only very local moves over the state space: moves that change one variable at a time. In models where variables are tightly correlated, such moves often lead from states whose probability is high to states whose probability is very low. In this case, the high-probability states will form strong basins of attraction, and the chain will be very unlikely to move away from such a state; that is, the chain will mix very slowly. In this case, we often want to consider chains that allow a broader range of moves, including much larger steps in the space.** The framework we develop in this section allows us to construct a broad family of chains in a way that guarantees the desired stationary distribution.

12.3.4.1 Detailed Balance

Before we address the question of how to construct a Markov chain with a particular stationary distribution, we address the question of how to verify easily that our Markov chain has the desired stationary distribution. Fortunately, we can define a test that is local and easy to check, and that suffices to characterize the stationary distribution. As we will see, this test also provides us with a simple method for constructing an appropriate chain.

Definition 12.5 *A finite-state Markov chain $\mathcal{T}$ is* reversible *if there exists a unique distribution π such that, for all $\boldsymbol{x}, \boldsymbol{x}' \in Val(\boldsymbol{X})$:*

reversible Markov chain

$$\pi(\boldsymbol{x})\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}') = \pi(\boldsymbol{x}')\mathcal{T}(\boldsymbol{x}' \to \boldsymbol{x}). \tag{12.24}$$

detailed balance

This equation is called the detailed balance. ■

The product $\pi(\boldsymbol{x})\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}')$ represents a process where we pick a starting state at random according to π, and then take a random transition from the chosen state according to the transition model. The detailed balance equation asserts that, using this process, the probability of a transition from $\boldsymbol{x}$ to $\boldsymbol{x}'$ is the same as the probability of a transition for $\boldsymbol{x}'$ to $\boldsymbol{x}$.

Reversibility implies that π is a stationary distribution of $\mathcal{T}$, but not necessarily that the chain will converge to π (see example 12.8). However, if $\mathcal{T}$ is regular, then convergence is guaranteed, and the reversibility condition provides a simple characterization of its stationary distribution:

Proposition 12.3 *If $\mathcal{T}$ is regular and it satisfies the detailed balance equation relative to π, then π is the unique stationary distribution of $\mathcal{T}$.*

The proof is left as an exercise (exercise 12.14).

Example 12.13 *We can test this proposition on the Markov chain of figure 12.4. Our detailed balance equation for the two states x^1 and x^3 asserts that*

$$\pi(x^1)\mathcal{T}(x^1 \to x^3) = \pi(x^3)\mathcal{T}(x^3 \to x^1).$$

Testing this equation for the stationary distribution π described in example 12.7, we have:

$$0.2 \cdot 0.75 = 0.3 \cdot 0.5 = 0.15.$$ ■

The detailed balance equation can also be applied to multiple kernels. If each kernel $\mathcal{T}_i$ satisfies the detailed balance equation relative to some stationary distribution π, then so does the mixture transition model $\mathcal{T}$ (see exercise 12.16). The application to the multistep transition model $\mathcal{T}$ is also possible, but requires some care (see exercise 12.17).

12.3.4.2 Metropolis-Hastings Algorithm

The reversibility condition gives us a condition for verifying that our Markov chain has the desired stationary distribution. However, it does not provide us with a constructive approach for producing such a Markov chain. The *Metropolis-Hastings algorithm* is a general construction that allows us to build a reversible Markov chain with a particular stationary distribution.

Metropolis-Hastings algorithm

Unlike the Gibbs chain, the algorithm does not assume that we can generate next-state samples from a particular target distribution. Rather, it uses the idea of a *proposal distribution* that we have already seen in the case of importance sampling.

proposal distribution

As for importance sampling, the proposal distribution in the Metropolis-Hastings algorithm is intended to deal with cases where we cannot sample directly from a desired distribution. In the case of a Markov chain, the target distribution is our next-state sampling distribution at a given state. We would like to deal with cases where we cannot sample directly from this target. Therefore, we sample from a different distribution — the proposal distribution — and then correct for the resulting error. However, unlike importance sampling, we do not want to keep track of importance weights, which are going to decay exponentially with the number of transitions, leading to a whole slew of problems. Therefore, we instead randomly choose whether to accept the proposed transition, with a probability that corrects for the discrepancy between the proposal distribution and the target.

More precisely, our proposal distribution $\mathcal{T}^Q$ defines a transition model over our state space: For each state $\boldsymbol{x}$, $\mathcal{T}^Q$ defines a distribution over possible successor states in $Val(\boldsymbol{X})$, from

which we select randomly a candidate next state $\boldsymbol{x}'$. We can either accept the proposal and transition to $\boldsymbol{x}'$, or reject it and stay at $\boldsymbol{x}$. Thus, for each pair of states $\boldsymbol{x}, \boldsymbol{x}'$ we have an *acceptance probability* $\mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}')$. The actual transition model of the Markov chain is then:

acceptance probability

$$\begin{array}{rcl} \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}') & = & \mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')\mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}') \qquad \boldsymbol{x} \neq \boldsymbol{x}' \\ \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}) & = & \mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}) + \sum_{\boldsymbol{x}' \neq \boldsymbol{x}} \mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')(1 - \mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}')). \end{array} \tag{12.25}$$

By using a proposal distribution, we allow the Metropolis-Hastings algorithm to be applied even in cases where we cannot directly sample from the desired next-state distribution; for example, where the distribution in equation (12.22) is too complex to represent. The choice of proposal distribution can be arbitrary, so long as it induces a regular chain. One simple choice in discrete factored state spaces is to use a multiple transition model, where $\mathcal{T}_i^Q$ is a uniform distribution over the values of the variable X_i.

Given a proposal distribution, we can use the detailed balance equation to select the acceptance probabilities so as to obtain the desired stationary distribution. For this Markov chain, the detailed balance equations assert that, for all $\boldsymbol{x} \neq \boldsymbol{x}'$,

$$\pi(\boldsymbol{x})\mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')\mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}') = \pi(\boldsymbol{x}')\mathcal{T}^Q(\boldsymbol{x}' \to \boldsymbol{x})\mathcal{A}(\boldsymbol{x}' \to \boldsymbol{x}).$$

We can verify that the following acceptance probabilities satisfy these equations:

$$\mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}') = \min\left[1, \frac{\pi(\boldsymbol{x}')\mathcal{T}^Q(\boldsymbol{x}' \to \boldsymbol{x})}{\pi(\boldsymbol{x})\mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')}\right], \tag{12.26}$$

and hence that the chain has the desired stationary distribution:

Theorem 12.5 *Let $\mathcal{T}^Q$ be any proposal distribution, and consider the Markov chain defined by equation (12.25) and equation (12.26). If this Markov chain is regular, then it has the stationary distribution π.*

The proof is not difficult, and is left as an exercise (exercise 12.15).

Let us see how this construction process works.

Example 12.14 *Assume that our proposal distribution $\mathcal{T}^Q$ is given by the chain of figure 12.4, but that we want to sample from a stationary distribution π' where: $\pi'(x^1) = 0.6$, $\pi'(x^2) = 0.3$, and $\pi'(x^3) = 0.1$. To define the chain, we need to compute the acceptance probabilities. Applying equation (12.26), we obtain, for example, that:*

$$\begin{array}{rcl} \mathcal{A}(x^1 \to x^3) & = & \min\left[1, \dfrac{\pi'(x^3)\mathcal{T}^Q(x^3 \to x^1)}{\pi'(x^1)\mathcal{T}^Q(x^1 \to x^3)}\right] = \min\left[1, \dfrac{0.1 \cdot 0.5}{0.6 \cdot 0.75}\right] = 0.11 \\ \mathcal{A}(x^3 \to x^1) & = & \min\left[1, \dfrac{\pi'(x^1)\mathcal{T}^Q(x^1 \to x^3)}{\pi'(x^3)\mathcal{T}^Q(x^3 \to x^1)}\right] = \min\left[1, \dfrac{0.6 \cdot 0.75}{0.1 \cdot 0.5}\right] = 1. \end{array}$$

We can now easily verify that the stationary distribution of the chain resulting from equation (12.25) and these acceptance probabilities gives the desired stationary distribution π'. ■

The Metropolis-Hastings algorithm has a particularly natural implementation in the context of graphical models. Each local transition model $\mathcal{T}_i$ is defined via an associated proposal

distribution $\mathcal{T}_i^{Q_i}$. The acceptance probability for this chain has the form

$$\begin{aligned}\mathcal{A}(\boldsymbol{x}_{-i}, x_i \to \boldsymbol{x}_{-i}, x_i') &= \min\left[1, \frac{\pi(\boldsymbol{x}_{-i}, x_i')\mathcal{T}_i^{Q_i}(\boldsymbol{x}_{-i}, x_i' \to \boldsymbol{x}_{-i}, x_i)}{\pi(\boldsymbol{x}_{-i}, x_i)\mathcal{T}_i^{Q_i}(\boldsymbol{x}_{-i}, x_i \to \boldsymbol{x}_{-i}, x_i')}\right] \\ &= \min\left[1, \frac{P_\Phi(x_i', \boldsymbol{x}_{-i})}{P_\Phi(x_i, \boldsymbol{x}_{-i})} \frac{\mathcal{T}_i^{Q_i}(\boldsymbol{x}_{-i}, x_i' \to \boldsymbol{x}_{-i}, x_i)}{\mathcal{T}_i^{Q_i}(\boldsymbol{x}_{-i}, x_i \to \boldsymbol{x}_{-i}, x_i')}\right].\end{aligned}$$

The proposal distributions are usually fairly simple, so it is easy to compute their ratios. In the case of graphical models, the first ratio can also be computed easily:

$$\begin{aligned}\frac{P_\Phi(x_i', \boldsymbol{x}_{-i})}{P_\Phi(x_i, \boldsymbol{x}_{-i})} &= \frac{P_\Phi(x_i' \mid \boldsymbol{x}_{-i})P_\Phi(\boldsymbol{x}_{-i})}{P_\Phi(x_i \mid \boldsymbol{x}_{-i})P_\Phi(\boldsymbol{x}_{-i})} \\ &= \frac{P_\Phi(x_i' \mid \boldsymbol{x}_{-i})}{P_\Phi(x_i \mid \boldsymbol{x}_{-i})}.\end{aligned}$$

As for Gibbs sampling, we can use the observation that each variable X_i is conditionally independent of the remaining variables in the network given its Markov blanket. Letting $\boldsymbol{U}_i$ denote $\mathrm{MB}_\mathcal{K}(X_i)$, and $\boldsymbol{u}_i = (\boldsymbol{x}_{-i})\langle \boldsymbol{U}_i \rangle$, we have that:

$$\frac{P_\Phi(x_i' \mid \boldsymbol{x}_{-i})}{P_\Phi(x_i \mid \boldsymbol{x}_{-i})} = \frac{P_\Phi(x_i' \mid \boldsymbol{u}_i)}{P_\Phi(x_i \mid \boldsymbol{u}_i)}.$$

This expression can be computed locally and efficiently, based only on the local parameterization of X_i and its Markov blanket (exercise 12.18).

The similarity to the derivation of Gibbs sampling is not accidental. Indeed, it is not difficult to show that Gibbs sampling is simply a special case of Metropolis-Hastings, one with a particular choice of proposal distribution (exercise 12.19).

The Metropolis-Hastings construction allows us to produce a Markov chain for an arbitrary stationary distribution. Importantly, however, we point out that the key theorem still requires that the constructed chain be regular. This property does not follow directly from the construction. In particular, the exclusive-or network of example 12.12 induces a nonregular Markov chain for any Metropolis-Hastings construction that uses a local proposal distribution — one that proposes changes to only a single variable at a time. In order to obtain a regular chain for this example, we would need a proposal distribution that allows simultaneous changes to both X and Y at a single step.

12.3.5 Using a Markov Chain

So far, we have discussed methods for defining Markov chains that induce the desired stationary distribution. Assume that we have constructed a chain that has a unique stationary distribution π, which is the one from which we wish to sample. How do we use this chain to answer queries? A naive answer is straightforward. We run the chain using the algorithm of algorithm 12.5 until it converges to the stationary distribution (or close to it). We then collect a sample from π. We repeat this process once for each particle we want to collect. The result is a data set $\mathcal{D}$ consisting of independent particles, each of which is sampled (approximately) from the stationary distribution π. The analysis of section 12.1 is applicable to this setting, so we can provide tight

bounds on the number of samples required to get estimators of a certain quality. Unfortunately, matters are not so straightforward, as we now discuss.

12.3.5.1 Mixing Time

burn-in time

A critical gap in this description of the MCMC algorithm is a specification of the *burn-in time* T — the number of steps we take until we collect a sample from the chain. Clearly, we want to wait until the state distribution is reasonably close to π. More precisely, we want to find a T that guarantees that, regardless of our starting distribution $P^{(0)}$, $P^{(T)}$ is within some small ϵ of π. In this context, we usually use variational distance (see section A.1.3.3) as our notion of "within ϵ."

Definition 12.6

Let $\mathcal{T}$ be a Markov chain. Let T_ϵ be the minimal T such that, for any starting distribution $P^{(0)}$, we have that:

$$\mathbf{D}_{\text{var}}(P^{(T)}; \pi) \leq \epsilon.$$

mixing time

Then T_ϵ is called the ϵ-mixing time *of $\mathcal{T}$.* ■

In certain cases, the mixing time can be extremely long. This situation arises in chains where the state space has several distinct regions each of which is well connected, but where transitions between regions are low probability. In particular, we can estimate the extent to which the chain allows mixing using the following quantity:

Definition 12.7

conductance

Let $\mathcal{T}$ be a Markov chain transition model and π its stationary distribution. The conductance *of $\mathcal{T}$ is defined as follows:*

$$\min_{\substack{\mathcal{S} \subset Val(\boldsymbol{X}) \\ 0 < \pi(\mathcal{S}) \leq 1/2}} \frac{P(\mathcal{S} \rightsquigarrow \mathcal{S}^c)}{\pi(\mathcal{S})},$$

where $\pi(\mathcal{S})$ is the probability assigned by the stationary distribution to the set of states $\mathcal{S}$, $\mathcal{S}^c = Val(\boldsymbol{X}) - \mathcal{S}$, and

$$P(\mathcal{S} \rightsquigarrow \mathcal{S}^c) = \sum_{\boldsymbol{x} \in \mathcal{S}, \boldsymbol{x}' \in \mathcal{S}^c} \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}').$$

■

Intuitively, $P(\mathcal{S} \rightsquigarrow \mathcal{S}^c)$ is the total "bandwidth" for transitioning from $\mathcal{S}$ to its complement. In cases where the conductance is low, there is some set of states $\mathcal{S}$ where, once in $\mathcal{S}$, it is very difficult to transition out of it. Figure 12.6 visualizes this type of situation, where the only transition between $\mathcal{S} = \{x^1, x^2, x^3\}$ and its complement is the dashed transition between x^2 and x^4, which has a very low probability. In cases such as this, if we start in a state within $\mathcal{S}$, the chain is likely to stay in $\mathcal{S}$ and to take a very long time before exploring other regions of the state space. Indeed, it is possible to provide both upper and lower bounds on the mixing rate of a Markov chain in terms of its conductance.

In the context of Markov chains corresponding to graphical models, chains with low conductance are most common in networks that have deterministic or highly skewed parameterization.

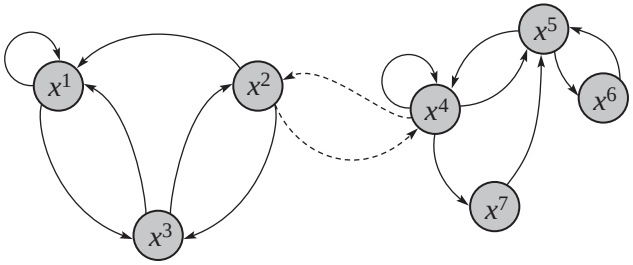


Figure 12.6 Visualization of a Markov chain with low conductance

In fact, as we saw in example 12.12, networks with deterministic CPDs might even lead to reducible chains, where different regions are entirely disconnected. However, even when the distribution is positive, we might still have regions that are connected only by very low-probability transitions. (See exercise 12.21.)

There are methods for providing tight bounds on the ϵ-mixing time of a given Markov chain. These methods are based on an analysis of the transition matrix between the states in the Markov chain.[3] Unfortunately, in the case of graphical models, an exhaustive enumeration of the exponentially many states is precisely what we wish to avoid. (If this enumeration were feasible, we would not have to resort to approximate inference techniques in the first place.) Alternatively, there is a suite of indirect techniques that allow us to provide bounds on the mixing time for some general class of chains. However, the application of these methods to each new class of chains requires a separate and usually quite sophisticated mathematical analysis. As of yet, there is no such analysis for the chains that are useful in the setting of graphical models. A more common approach is to use a variety of heuristics to try to evaluate the extent to which a sample trajectory has "mixed." See box 12.B for some further discussion.

12.3.5.2 Collecting Samples

The burn-in time for a large Markov chain is often quite large. Thus, the naive algorithm described above has to execute a large number of sampling steps for every usable sample. However, a key observation is that, if $\boldsymbol{x}^{(t)}$ is sampled from π, then $\boldsymbol{x}^{(t+1)}$ is also sampled from π. Thus, once we have run the chain long enough that we are sampling from the stationary distribution (or a distribution close to it), we can continue generating samples from the same trajectory and obtain a large number of samples from the stationary distribution.

More formally, assume that we use $\boldsymbol{x}^{(0)}, \ldots, \boldsymbol{x}^{(T)}$ as our burn-in phase, and then collect M samples $\mathcal{D} = \{\boldsymbol{x}[1], \ldots, \boldsymbol{x}[M]\}$ from the stationary distribution. Most simply, we might collect M consecutive samples, so that $\boldsymbol{x}[m] = \boldsymbol{x}^{(T+m)}$, for $m = 1, \ldots, M$. If $\boldsymbol{x}^{(T+1)}$ is sampled from π, then so are all of the samples in $\mathcal{D}$. Thus, if our chain has mixed by the time we collect

3. Specifically, they involve computing the second largest eigen-value of the matrix.

our first sample, then for any function f,

$$\hat{\boldsymbol{E}}_{\mathcal{D}}(f) = \frac{1}{M}\sum_{m=1}^{M} f(\boldsymbol{x}[m], \boldsymbol{e})$$

estimator

is an unbiased *estimator* for $\boldsymbol{E}_{\pi(\boldsymbol{X})}[f(\boldsymbol{X}, \boldsymbol{e})]$.

How good is this estimator? As we discussed in appendix A.2.1, the quality of an unbiased estimator is measured by its variance: the lower the variance, the higher the probability that the estimator is close to its mean. In theorem A.2, we showed an analysis of the variance of an estimator obtained from M independent samples. Unfortunately, we cannot apply that analysis in this setting. The key problem, of course, is that consecutive samples from the same trajectory are correlated. Thus, we cannot expect the same performance as we would from M independent samples from π. More formally, the variance of the estimator is significantly higher than that of an estimator generated by M independent samples from π, as discussed before.

Example 12.15

Consider the Gibbs chain for the deterministic exclusive-or network of example 12.12, and assume we compute, for a given run of the chain, the fraction of states in which x^1 holds in the last 100 states traversed by the chain. A chain started in the state x^1, y^0 would have that $100/100$ of the states have x^1, whereas a chain started in the state x^0, y^1 would have that $0/100$ of the states have x^1. Thus, the variance of the estimator is very high in this case. ■

central limit theorem

One can formalize this intuition by the following generalization of the *central limit theorem* that applies to samples collected from a Markov chain:

Theorem 12.6

Let $\mathcal{T}$ be a Markov chain and $\boldsymbol{X}[1], \ldots, \boldsymbol{X}[M]$ a set of samples collected from $\mathcal{T}$ at its stationary distribution P. Then, since $M \longrightarrow \infty$:

$$\left(\hat{\boldsymbol{E}}_{\mathcal{D}}(f) - \boldsymbol{E}_{\boldsymbol{X}\sim P}[f(\boldsymbol{X})]\right) \longrightarrow \mathcal{N}\left(0; \sigma_f^2\right)$$

where

$$\sigma_f^2 = \mathbb{V}ar_{\boldsymbol{X}\sim\mathcal{T}}[f(\boldsymbol{X})] + 2\sum_{\ell=1}^{\infty} \mathbb{C}ov_{\mathcal{T}}[f(\boldsymbol{X}[m]); f(\boldsymbol{X}[m+\ell])] < \infty.$$

autocovariance

The terms in the summation are called *autocovariance* terms, since they measure the covariance between samples from the chain, taken at different lags. The stronger the correlations between different samples, the larger the autocovariance terms, the higher the variance of our estimator. This result is consistent with the behavior we discussed in example 12.12.

We want to use theorem 12.6 in order to assess the quality of our estimator. In order to do so, we need to estimate the quantity σ_f^2. We can estimate the variance from our empirical data using the standard estimator:

$$\mathbb{V}ar_{\boldsymbol{X}\sim\mathcal{T}}[f(\boldsymbol{X})] \approx \frac{1}{M-1}\left[\sum_{m=1}^{M}\left(f(\boldsymbol{X}) - \hat{\boldsymbol{E}}_{\mathcal{D}}(f)\right)^2\right]. \tag{12.27}$$

To estimate the autocovariance terms from the empirical data, we compute:

$$\mathbb{C}ov_{\mathcal{T}}[f(\boldsymbol{X}[m]); f(\boldsymbol{X}[m+\ell])] \approx \frac{1}{M-\ell}\sum_{m=1}^{M-\ell}(f(\boldsymbol{X}[m] - \hat{\boldsymbol{E}}_{\mathcal{D}}(f))(f(\boldsymbol{X}[m+\ell] - \hat{\boldsymbol{E}}_{\mathcal{D}}(f)).$$

(12.28)

At first glance, theorem 12.6 suggests that the variance of the estimate could be reduced if the chain is allowed a sufficient number of iterations between sample collections. Thus, having collected a particle $\boldsymbol{x}^{(T)}$, we can let the chain run for a while, and collect a second particle $\boldsymbol{x}^{(T+d)}$ for some appropriate choice of d. For d large enough, $\boldsymbol{x}^{(T)}$ and $\boldsymbol{x}^{(T+d)}$ are only slightly correlated, reducing the correlation in the preceding theorem.

However, this approach is suboptimal for various reasons. First, the time d required for "forgetting" the correlation is clearly related to the mixing time of the chain. Thus, chains that are slow to mix initially also require larger d in order to produce close-to-independent particles. Nevertheless, the samples do come from the correct distribution for any value of d, and hence it is often better to compromise and use a shorter d than it is to use a shorter burn-in time T. This method thus allows us to collect a larger number of usable particles with fewer transitions of the Markov chain. Indeed, **although the samples between $\boldsymbol{x}^{(T)}$ and $\boldsymbol{x}^{(T+d)}$ are not independent samples, there is no reason to discard them. That is, one can show that using all of the samples $\boldsymbol{x}^{(T)}, \boldsymbol{x}^{(T+1)}, \ldots, \boldsymbol{x}^{(T+d)}$ produces a provably better estimator than using just the two samples $\boldsymbol{x}^{(T)}$ and $\boldsymbol{x}^{(T+d)}$: our variance is always no higher if we use all of the samples we generated rather than a subset. Thus, the strategy of picking only a subset of the samples is useful primarily in settings where there is a significant cost associated with using each sample (for example, the evaluation of f is costly), so that we might want to reduce the overall number of particles used.**

Box 12.B — Skill: MCMC in Practice. *A key question when using a Markov chain is evaluating the time required for the chain to "mix" — that is, approach the stationary distribution. As we discussed, no general-purpose theoretical analysis exists for the mixing time of graphical models. However, we can still hope to estimate the extent to which a sample trajectory has "forgotten" its origin. Recall that, as we discussed, the most common problem with mixing arises when the state space consists of several regions that are connected only by low-probability transitions. If we start the chain in a state in one of these regions, it is likely to spend some amount of time in that same region before transitioning to another region. Intuitively, the states sampled in the initial phase are clearly not from the stationary distribution, since they are strongly correlated with our initial state, which is arbitrary. However, later in the trajectory, we might reach a state where the current state is as likely to have originated in any initial state. In this case, we might consider the chain to have mixed.*

Diagnosing convergence of a Markov chain Monte Carlo method is a notoriously hard problem. The chain may appear to have converged simply by spending a large number of iterations in a particular mode due to low conductance between modes. However, there are approaches that can tell us if a chain has not converged.

One technique is based directly on theorem 12.6. In particular, we can compute the ratio ρ_ℓ of the estimated autocovariance in equation (12.28) to the estimated variance in equation (12.27). This ratio is known as the autocorrelation *of lag ℓ; it provides a normalized estimate of the extent to which the chain has mixed in ℓ steps. In practice, the autocorrelation should drop off exponentially with the length of the lag, and one way to diagnose a poorly mixing chain is to observe high autocorrelation at distant lags. Note, however, that the number of samples available for computing autocorrelation decreases with lag, leading to large variance in the autocorrelation estimates at large lags.*

A different technique uses the observation that multiple chains sampling the same distribution should, upon convergence, all yield similar estimates. In addition, estimates based on a complete set of samples collected from all of the chains should have variance comparable to variance in each of the chains. More formally, assume that K separate chains are each run for $T + M$ steps starting from a diverse set of starting points. After discarding the first T samples from each chain, let $\boldsymbol{X}_k[m]$ denote a sample from chain k after iteration $T + m$. We can now compute the B (between-chains) and W (within-chain) variances:

$$\begin{aligned}
\bar{f}_k &= \frac{1}{M}\sum_{m=1}^{M} f(\boldsymbol{X}_k[m]) \\
\bar{f} &= \frac{1}{K}\sum_{k=1}^{K} \bar{f}_k \\
B &= \frac{M}{K-1}\sum_{k=1}^{K}(\bar{f}_k - \bar{f})^2 \\
W &= \frac{1}{K}\frac{1}{M-1}\sum_{k=1}^{K}\sum_{m=1}^{M}\left(f(\boldsymbol{X}_k[m]) - \bar{f}_k\right)^2 .
\end{aligned}$$

The expression $V = \frac{M-1}{M}W + \frac{1}{M}B$ can now be shown to overestimate the variance of our estimate of f based on the collected samples. In the limit of $M \longrightarrow \infty$, both W and V converge to the true variance of the estimate. One measure of disagreement between chains is given by $\hat{R} = \sqrt{\frac{V}{W}}$. If the chains have not all converged to the stationary distribution, this estimate will be high. If this value is close to 1, either the chains have all converged to the true distribution, or the starting points were not sufficiently dispersed and all of the chains have converged to the same mode or a set of modes. We can use this strategy with multiple different functions f in order to increase our confidence that our chain has mixed. We can, for example, use indicator functions of various events, as well as more complex functions of multiple variables.

Overall, although the strategy of using only a single chain produces more viable particles using lower computational cost, there are still significant advantages to the multichain approach. First, by starting out in very different regions of the space, we are more likely to explore a more representative subset of states. Second, the use of multiple chains allows us to evaluate the extent to which our chains are mixing. Thus, **to summarize, a good strategy for using a Markov chain in practice is a hybrid approach, where we run a small number of chains in parallel for a reasonably long time, using their behavior to evaluate mixing. After the burn-in phase, we then use the existence of multiple chains to estimate convergence. If mixing appears to have occurred, we can use each of our chains to generate multiple particles, remembering that the particles generated in this fashion are not independent.**

12.3.5.3 Discussion

MCMC methods have many advantages over other methods. Unlike the global approximate inference methods of the previous chapter, they can, at least in principle, get arbitrarily close

to the true posterior. Unlike forward sampling methods, these methods do not degrade when the probability of the evidence is low, or when the posterior is very different from the prior. Furthermore, unlike forward sampling, MCMC methods apply to undirected models as well as to directed models. As such, they are an important component in the suite of approximate inference techniques.

However, MCMC methods are not generally an out-of-the-box solution for dealing with inference in complex models. First, the application of MCMC methods leaves many options that need to be specified: the proposal distribution, the number of chains to run, the metrics for evaluating mixing, techniques for determining the delay between samples that would allow them to be considered independent, and more. Unfortunately, at this point, there is little theoretical analysis that can help answer these questions for the chains that are of interest to us. Thus, the application of Markov chains is more of an art than a science, and it often requires significant experimentation and hand-tuning of parameters.

Second, MCMC methods are only viable if the chain we are using mixes reasonably quickly. Unfortunately, many of the chains derived from real-world graphical models frequently have multimodal posterior distributions, with slow mixing between the modes. For such chains, the straightforward MCMC methods described in this chapter are unlikely to work. In such cases, diagnostics such as the ones described in box 12.B can be used to determine that the chain is not mixing, and better methods must then be applied. **The key to improving the convergence of a Markov chain is to introduce transitions that take larger steps in the space, allowing the chain to move more rapidly between modes, and thereby to better explore the space. The best strategy is often to analyze the properties of the posterior landscape of interest, and to construct moves that are tailored for this specific space. (See, for example, exercise 12.23.) Fortunately, the ability to mix different reversible kernels within a single chain (as discussed in section 12.3.4) allows us to introduce a variety of long-range moves while still maintaining the same target posterior.**

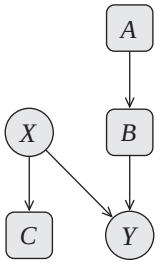
In addition to the use of long-range steps that are specifically designed for particular (classes of) chains, there are also some general-purpose methods that try to achieve that goal. The block Gibbs approach (section 12.3.3) is an instance of this general class of methods. Another strategy uses the same ideas in *simulated annealing* to improve convergence of local search to a better optimum. Here, we can define an intermediate distribution parameterized by a *temperature parameter* T: T:

simulated annealing

temperature parameter

$$\tilde{P}_T(\boldsymbol{X}) \propto \exp\{-\frac{1}{T}\log \tilde{P}(\boldsymbol{X})\}.$$

This distribution is similar to our original target distribution $\tilde{P}$. At a low temperature of $T = 1$, this equation yields the original target distribution. But as the temperature increases, modes become broader and merge, reducing the multimodality of the distribution and increasing its mixing rate. We can now define various methods that use a combination of related chains running at different temperatures. At a high level, the higher-temperature chain can be viewed as proposing a step, which we can accept or reject using the acceptance probability of our true target distribution. (See section 12.7 for references to some of these more advanced methods.) In effect, these approaches use the higher-temperature chains to define a set of larger steps in the space, thereby providing a general-purpose method for achieving more rapid movement between multiple modes. However, this generality comes at the computational cost of running parallel



(a)

```
var
   A, B, C, X, Y, mu, tau, p[2,3], q;

p = ...
A ~ dbern(0.3)
B ~ dcat(p[A,1:3])
X ~ dnorm(-1,0.25)
mu <- 3*X+B^2
tau <- 1/X^2
Y ~ dnorm(mu,tau)
logit(q) <- 4*X + 2
C ~ dbern(q)
```

(b)

Figure 12.C.1 — Example of BUGS model specification (a) A simple hybrid Bayesian network. (b) A BUGS definition of a probabilistic model over this network.

chains; thus, if we can understand our specific posterior well enough to construct specialized operators that move between modes, that often provides a more effective solution.

Box 12.C — Case Study: The BUGS System. *One of the main advantages of MCMC methods is their broad applicability to a very general class of networks. Not only do they apply (at least in principle) to any discrete network, regardless of its complexity, they also generalize fairly simply to continuous variables (see section 14.5.3). One very useful system that exploits this generality is the* BUGS *system, developed by Thomas et al. (1992). This system provides a general-purpose language for representing a broad range of probabilistic models and uses MCMC to run inference over these models.*

BUGS system

The BUGS system *provides a programming-language-based representation of a probabilistic model. The model defines a joint distribution over a set of random variables. Variables can be defined as functions of each other; these functions can be deterministic functions, or stochastic functions utilizing a rich set of predefined distributions. For example, consider the simple Bayesian network shown in figure 12.C.1a, where A, B, C are discrete and X, Y are continuous. One possible probabilistic model can be written in* BUGS *using the commands shown in figure 12.C.1b. This model defines: A to be a binary-valued variable, with $P(a^1) = 0.3$; B is a 3-valued variable that depends on A, whose CPT is defined in the matrix P; X is a Gaussian random variable with mean -1 and precision (inverse variance) 0.25; Y is a conditional Gaussian whose mean depends on X and B and whose precision also depends on X; and C is a logistic function of $4X + 2$. Even in this very simple example, we can see that the* BUGS *language provides a rich language for encoding different families of functional and stochastic dependencies between variables.*

Given a probabilistic model defined in this way, the BUGS *system can instantiate evidence for some*

of the variables (for example, by reading their values from a file) and then perform inference over the model by running various MCMC algorithms. The system analyzes the parametric form specifying the distribution of the different variables, and it selects an appropriate sampling algorithm to use. The user specifies the number of sampling iterations to perform, and which variables are to be monitored *— their values are to be stored during the MCMC iterations. We can then compute such values as the mean and standard deviation of these monitored variables. The system also provides various methods to help detect convergence of the MCMC runs (see also box 12.B).*

Overall, the BUGS *tool provides a general-purpose and highly flexible framework for specifying and reasoning with probabilistic models. Its ability to provide such a high level of expression power rests on the generality of MCMC as an inference method, and its applicability to a very broad range of distributions (broader than any other inference method currently available).*

12.4 Collapsed Particles

So far, we have restricted our attention to methods that use as their particles only instantiations ξ to all the network variables. Clearly, covering an exponentially large state space with a small number of instantiations is difficult, and it often takes a large number of full particles to obtain reasonable estimates. One approach for improving the performance of particle-based methods is to use as particles *partial* assignments to some subset of the network variables, combined with some type of closed-form representation of a distribution over the rest.

More precisely, assume that we partition $\mathcal{X}$ into two subsets: $\boldsymbol{X}_p$ — the variables whose assignment defines the particle, and $\boldsymbol{X}_d$ — the variables over which we will maintain a closed-form distribution. Then *collapsed particles* consist of an instantiation $\boldsymbol{x}_p \in Val(\boldsymbol{X}_p)$, coupled with some representation of the distribution $P(\boldsymbol{X}_d \mid \boldsymbol{x}_p, \boldsymbol{e})$. The particle is "collapsed" because some of the variables are not assigned but rather summarized using a distribution. Collapsed particles are also known as *Rao-Blackwellized particles*.

collapsed particles

Assume that we want to estimate an expectation of some function $f(\xi)$ relative to our posterior distribution $P(\boldsymbol{X}_p, \boldsymbol{X}_d \mid \boldsymbol{e})$. We now have:

$$\begin{aligned} \boldsymbol{E}_{P(\xi|\boldsymbol{e})}[f(\xi)] &= \sum_{\boldsymbol{x}_p, \boldsymbol{x}_d} P(\boldsymbol{x}_p, \boldsymbol{x}_d \mid \boldsymbol{e}) f(\boldsymbol{x}_p, \boldsymbol{x}_d, \boldsymbol{e}) \\ &= \sum_{\boldsymbol{x}_p} P(\boldsymbol{x}_p \mid \boldsymbol{e}) \sum_{\boldsymbol{x}_d} P(\boldsymbol{x}_d \mid \boldsymbol{x}_p, \boldsymbol{e}) f(\boldsymbol{x}_p, \boldsymbol{x}_d, \boldsymbol{e}) \\ &= \sum_{\boldsymbol{x}_p} P(\boldsymbol{x}_p \mid \boldsymbol{e}) \left(\boldsymbol{E}_{P(\boldsymbol{X}_d|\boldsymbol{x}_p,\boldsymbol{e})}[f(\boldsymbol{x}_p, \boldsymbol{X}_d, \boldsymbol{e})] \right). \end{aligned} \tag{12.29}$$

We can use the samples $\boldsymbol{x}_p[m]$ to approximate any expectation relative to the distribution $P(\boldsymbol{X}_p \mid \boldsymbol{e})$, using the techniques described above. In particular, we can approximate the expectation of the expression in parentheses — an expression that is itself an expectation.

In the case of collapsed particles, we assume that the internal expectation can be computed (or approximated) efficiently. As we will discuss, we can explicitly represent the distribution $P(\boldsymbol{X}_d \mid \boldsymbol{x}_p, \boldsymbol{e})$ as a graphical model, using constructions such as the reduced Markov network of section 4.2.3. We thus have a hybrid approach where we generate samples $\boldsymbol{x}_p$ from $\boldsymbol{X}_p$

and perform exact inference on $\boldsymbol{X}_d$ given $\boldsymbol{x}_p$. Thus, this approach defines a spectrum: When $\boldsymbol{X}_p = \boldsymbol{X}$, collapsed particles are simply full particles, and we are simply applying the methods defined earlier in this chapter; when $\boldsymbol{X}_p = \emptyset$, we have a single particle whose associated distribution is our original network, so that we are back in the regime of exact inference. We note that, in some cases, the network might not be sufficiently simple for exact inference. However, it might be amenable to some other form of approximation — for example, one of the methods we discuss in chapter 11. **Intuitively, the fewer variables we sample (keep in $\boldsymbol{X}_p$), the larger the part of the probability mass that we cover using each collapsed particle $\boldsymbol{x}_p[m]$. From an alternative perspective, we are performing an exact computation for the expectation relative to $\boldsymbol{X}_d$, thereby eliminating any contribution it makes to the bias or the variance of the estimator. Thus, if $|\boldsymbol{X}_p|$ is fairly small, we can obtain much better estimates for the distribution using significantly fewer particles.**

In this section, we describe extensions of the approaches discussed earlier in this chapter to the case of collapsed particles.

12.4.1 Collapsed Likelihood Weighting ⋆

We begin by describing a collapsed extension to likelihood weighting. We first describe the algorithm generally, from the perspective of normalized importance sampling. We then consider a specific application that is a direct extension to the full-particle version of likelihood weighting.

12.4.1.1 Collapsed Importance Sampling

Recall that, in importance sampling, we generate our samples from an alternative proposal distribution Q, and we compensate for the discrepancy by associating with each particle a weight $w[m]$. In the case of collapsed particles, we are generating specific particles only for the variables in $\boldsymbol{X}_p$, so that Q would be a distribution over $\boldsymbol{x}_p$. We thus generate a data set

$$\mathcal{D} = \{(\boldsymbol{x}_p[m], w[m], P(\boldsymbol{X}_d \mid \boldsymbol{x}_p[m], \boldsymbol{e}))\}_{m=1}^{M},$$

where each $\boldsymbol{x}_p[m]$ is sampled from Q. Here we will discuss both the choice of proposal distribution Q and the computation of the weights $w[m]$.

Our goal is to estimate the expectation of equation (12.29). In effect, we are estimating the expectation of a new function g, which represents the internal expectation: $g = \boldsymbol{E}_{P(\boldsymbol{X}_d|\boldsymbol{x}_p,\boldsymbol{e})}[f(\boldsymbol{x}_p, \boldsymbol{X}_d, \boldsymbol{e})]$. Using normalized importance sampling, we estimate this expectation as:

$$\hat{\boldsymbol{E}}_{\mathcal{D}}(f) = \frac{\sum_{m=1}^{M} w[m] \left(\boldsymbol{E}_{P(\boldsymbol{X}_d|\boldsymbol{x}_p[m],\boldsymbol{e})}[f(\boldsymbol{x}_p[m], \boldsymbol{X}_d, \boldsymbol{e})]\right)}{\sum_{m=1}^{M} w[m]}. \tag{12.30}$$

Example 12.16 *Consider the* Extended Student *network, repeated in figure 12.7a, with the evidence d^1, h^0. Assume that we choose to partition the variables as follows: $\boldsymbol{X}_p = \{D, G\}$, and $\boldsymbol{X}_d = \{C, I, S, L, J, H\}$. In this case, each particle defines an assignment (d, g) to the variables D, G. Assuming that our algorithm follows the template of full-particle likelihood weighting, we would ensure that our proposal distribution Q ascribes only positive probability to particles (d^1, g) that are compatible with our evidence d^1. Each such particle is also associated with a distribution $P(C, I, S, L, H \mid g, d^1, h^0)$. The reduced Markov network shown in figure 12.7b represents this distribution.*

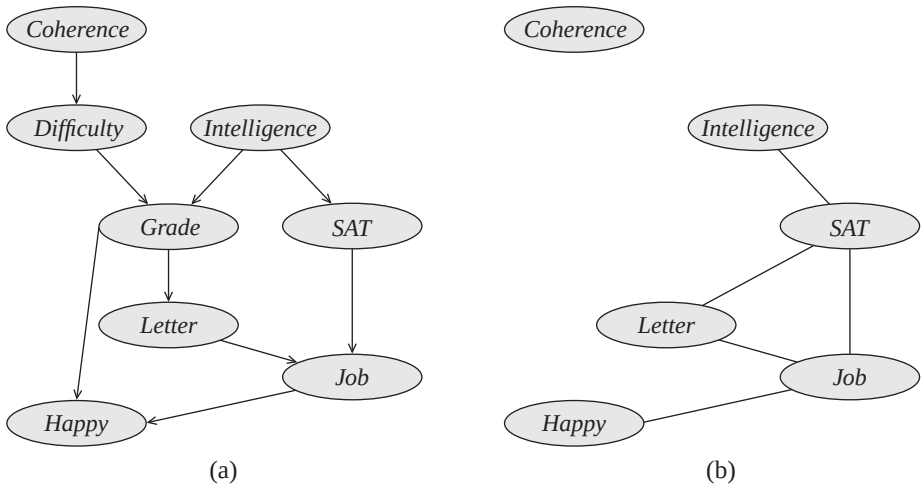


Figure 12.7 Networks illustrating collapsed importance sampling: (a) The Extended-Student Bayesian network $\mathcal{B}^{student}$; (b) The network $\mathcal{B}^{student}_{G=g,D=d}$ reduced by $G = g, D = d$.

Now, assume that our query is $P(j^1 \mid d^1, h^0)$, so that our function f is the indicator function $\mathbf{1}\{\xi\langle J\rangle = j^1\}$. For each particle, we evaluate

$$\mathbf{E}_{P(C,I,S,L,J,H|g,d^1,h^0)}\left[\mathbf{1}\{\xi\langle J\rangle = j^1\}\right] = P(j^1 \mid g, d^1, h^0).$$

We then compute an average of these probabilities, weighted by the importance weights (which we will discuss). The computation of the probabilities $P(J \mid g, d^1, h^0)$ can be done using inference in the reduced network, which is simpler than inference in the original network. (Although, in this very simple network, the savings are not significant.)

Note that the extent to which the reduced network allows more effective inference than the original one depends on our choice of variables $\boldsymbol{X}_p$. For example, if (for some reason) we choose $\boldsymbol{X}_p = \{H\}$, the resulting conditioned network is no simpler than the original, and the use of particle-based methods has no computational benefits over the use of exact inference. ■

12.4.1.2 Formal Description

To specify the algorithm formally, we must define the proposal distribution Q and the associated importance weights. We begin by partitioning our evidence set $\boldsymbol{E}$ into two subsets: $\boldsymbol{E}_p = \boldsymbol{E} \cap \boldsymbol{X}_p$, and $\boldsymbol{E}_d = \boldsymbol{E} \cap \boldsymbol{X}_d$, with $\boldsymbol{e}_p$ and $\boldsymbol{e}_d$ defined accordingly. This partition determines how we handle each of the observed variables: evidence in $\boldsymbol{E}_p$ will be treated as in likelihood weighting, modifying our sampling process and the importance weights; evidence in $\boldsymbol{E}_d$ will be accounted for as part of the exact inference process.

Now, consider an arbitrary proposal distribution Q. We can go through an analysis similar to

the one that allowed us to derive equation (12.12):

$$\begin{aligned}\boldsymbol{E}_{P(\xi|\boldsymbol{e})}[f(\xi)] &= \sum_{\boldsymbol{x}_p,\boldsymbol{x}_d} P(\boldsymbol{x}_p,\boldsymbol{x}_d \mid \boldsymbol{e}) f(\boldsymbol{x}_p,\boldsymbol{x}_d,\boldsymbol{e}) \\ &= \sum_{\boldsymbol{x}_p} Q(\boldsymbol{x}_p)\frac{P(\boldsymbol{x}_p \mid \boldsymbol{e})}{Q(\boldsymbol{x}_p)} \sum_{\boldsymbol{x}_d} P(\boldsymbol{x}_d \mid \boldsymbol{x}_p,\boldsymbol{e}) f(\boldsymbol{x}_p,\boldsymbol{x}_d,\boldsymbol{e}).\end{aligned}$$

We can now reformulate the term $P(\boldsymbol{x}_p \mid \boldsymbol{e})$ as:

$$\begin{aligned}P(\boldsymbol{x}_p \mid \boldsymbol{e}) &= \frac{P(\boldsymbol{x}_p,\boldsymbol{e})}{P(\boldsymbol{e})} \\ &= \frac{P(\boldsymbol{x}_p,\boldsymbol{e}_p,\boldsymbol{e}_d)}{P(\boldsymbol{e})} \\ &= \frac{1}{P(\boldsymbol{e})} P(\boldsymbol{x}_p,\boldsymbol{e}_p) P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p).\end{aligned}$$

Plugging this result back into our derivation, we obtain that:

$$\begin{aligned}\boldsymbol{E}_{P(\xi|\boldsymbol{e})}[f(\xi)] &= \frac{1}{P(\boldsymbol{e})} \sum_{\boldsymbol{x}_p} Q(\boldsymbol{x}_p)\frac{P(\boldsymbol{x}_p,\boldsymbol{e}_p)}{Q(\boldsymbol{x}_p)} P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p) \sum_{\boldsymbol{x}_d} P(\boldsymbol{x}_d \mid \boldsymbol{x}_p,\boldsymbol{e}) f(\boldsymbol{x}_p,\boldsymbol{x}_d,\boldsymbol{e}) \\ &= \frac{1}{P(\boldsymbol{e})} \boldsymbol{E}_{Q(\boldsymbol{X}_p)}\left[\frac{P(\boldsymbol{x}_p,\boldsymbol{e}_p)}{Q(\boldsymbol{x}_p)} P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p) \boldsymbol{E}_{P(\boldsymbol{x}_d|\boldsymbol{x}_p,\boldsymbol{e})}[f(\boldsymbol{x}_p,\boldsymbol{x}_d,\boldsymbol{e})]\right]. \qquad (12.31)\end{aligned}$$

This analysis suggests that the appropriate importance weights should be defined as:

$$w(\boldsymbol{x}_p) = \frac{P(\boldsymbol{x}_p,\boldsymbol{e}_p)}{Q(\boldsymbol{x}_p)} P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p). \qquad (12.32)$$

Indeed, if we compute the mean of our importance weights, as in equation (12.11), we obtain the following formula for the normalized importance sampling estimator:

$$\begin{aligned}\boldsymbol{E}_{Q(\boldsymbol{X}_p)}[w(\boldsymbol{X}_p)] &= \sum_{\boldsymbol{x}_p} Q(\boldsymbol{x}_p)\frac{P(\boldsymbol{x}_p,\boldsymbol{e}_p)}{Q(\boldsymbol{x}_p)} P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p) \\ &= \sum_{\boldsymbol{x}_p} P(\boldsymbol{x}_p,\boldsymbol{e}_p) P(\boldsymbol{e}_d \mid \boldsymbol{x}_p,\boldsymbol{e}_p) \\ &= \sum_{\boldsymbol{x}_p} P(\boldsymbol{e}_d,\boldsymbol{x}_p,\boldsymbol{e}_p) = P(\boldsymbol{e}_d,\boldsymbol{e}_p).\end{aligned}$$

Thus, if we select our importance weights as in equation (12.32), we have that:

$$\boldsymbol{E}_{P(\xi|\boldsymbol{e})}[f(\xi)] = \frac{\boldsymbol{E}_{Q(\boldsymbol{X}_p)}\left[w(\boldsymbol{X}_p)\boldsymbol{E}_{P(\boldsymbol{x}_d|\boldsymbol{x}_p,\boldsymbol{e})}[f(\boldsymbol{x}_p,\boldsymbol{x}_d,\boldsymbol{e})]\right]}{\boldsymbol{E}_{Q(\boldsymbol{X}_p)}[w(\boldsymbol{X}_p)]},$$

as desired.

12.4.1.3 Proposal Distribution

Our preceding analysis does not place any restrictions on the proposal distribution; we can choose any proposal distribution Q that seems appropriate (as long as it dominates P). However, it is important to remember our two main desiderata for a proposal distribution: easy generation of samples from Q and similarity between Q and our target distribution $P(\boldsymbol{X}_p \mid \boldsymbol{e})$. The proposal distribution we used for the full particle case attempted to address both of these desiderata, at least to some extent. In this section, we describe a generalization of that proposal distribution for the case of collapsed particles.

Our goal is to generate particles from a distribution $Q(\boldsymbol{X}_p)$. Following the template for the full particle case, we would sample each unobserved variable $X \in \boldsymbol{X}_p$ from its CPD. The reason we can execute this process is that we were careful to sample the parents of X_i before that, so that the distribution from which we should sample X is uniquely defined. In the collapsed case, however, Pa_X might not be within the set $\boldsymbol{X}_p$, in which case we would not have values for them when sampling X. For example, returning to the setting of example 12.16, it is not clear how we would define a sampling distribution for the variable G.

Most simply, we can select as our subset $\boldsymbol{X}_p$ an upwardly closed subset of nodes in the network. In our example, we might select $\boldsymbol{X}_p$ to consist of all of the variables C, D, I, G. This variant of the algorithm is very close to full-particle likelihood weighting. Here, we are back in a situation where we can order the variables in such a way that each unobserved variable can be sampled using its original CPD, using the previously sampled assignment to its parents. Each observed variable X_i is "sampled," as in standard likelihood weighting, from a mutilated network that ascribes probability 1 to its observed value $\boldsymbol{e}_p\langle X_i\rangle$. The computation of the importance weights for this case is straightforward: We first compute the part of the importance weight corresponding to $P(\boldsymbol{x}_p, \boldsymbol{e}_p)/Q(\boldsymbol{x}_p)$, using precisely the same incremental computation as in standard likelihood weighting. We then compute $P(\boldsymbol{e}_d \mid \boldsymbol{x}_p, \boldsymbol{e}_p)$ in the network conditioned on $\boldsymbol{x}_p, \boldsymbol{e}_p$, and we multiply the importance weight by this additional factor.

Example 12.17

Continuing example 12.16, assume we choose $\boldsymbol{X}_p$ to be the upwardly closed set C, D, I, G. We would sample C, I, and G from their CPD; D would be sampled from a CPD that has no parents, and ascribes probability 1 to d^1. The importance weight for a particle (c, d^1, i, g) is then computed as $P(d^1 \mid c) \cdot P(h^0 \mid c, d^1, i, g)$. Note that this last term requires inference in the network, specifically, the marginalization of L, J, S. ■

More generally, we can define a fully general proposal distribution Q by specifying a topological ordering $X_1, \ldots, X_k$ over $\boldsymbol{X}_p$, and a proposal distribution defined in terms of a Bayesian network $\mathcal{G}_Q$ over $\boldsymbol{X}_p$ that specifies, for each variable X_i, $i = 1, \ldots, k$, a parent set $\mathrm{Pa}_{X_i}^{\mathcal{G}_Q} \subseteq \{X_1, \ldots, X_{i-1}\}$, and a CPD $Q(X_i \mid \mathrm{Pa}_{X_i}^{\mathcal{G}_Q})$. This approach allows us to represent an arbitrary proposal distribution.

We can now consider the computation of each of the three terms in the definition of the importance weights in equation (12.32). The term $Q(\boldsymbol{x}_p)$ can be computed very simply via the chain rule for the proposal network $\mathcal{G}_Q$. The terms $P(\boldsymbol{e}_d \mid \boldsymbol{x}_p, \boldsymbol{e}_p)$ can be computed using inference in the conditioned network $\mathcal{B}_{\boldsymbol{E}_p=\boldsymbol{e}_p, \boldsymbol{X}_p=\boldsymbol{x}_p}$ — we compute the probability of the query $P(\boldsymbol{e}_d \mid \boldsymbol{e}_p, \boldsymbol{x}_p)$ in this network. As we stated, the whole approach of collapsed particles is based on the premise that (exact or approximate) inference in this conditioned network is feasible. Similarly, we can compute $P(\boldsymbol{x}_p, \boldsymbol{e}_p)$ in the same network.

12.4.2 Collapsed MCMC

The collapsed MCMC algorithm is also based on equation (12.29); however, as in section 12.3, we simplify our notation by defining Φ to be the set of factors reduced by the evidence $\boldsymbol{e}$, so that $P_\Phi(\boldsymbol{X}) = P(\boldsymbol{X} \mid \boldsymbol{e})$ (for $\boldsymbol{X} = \mathcal{X} - \boldsymbol{E}$). As in collapsed likelihood weighting, we approximate the outer expectation by generating particles that are instantiations of $\boldsymbol{X}_p$. Here, we generate the particles $\boldsymbol{x}_p[m]$ using a Markov chain process; at the limit of the chain, these particles will be sampled from the marginal posterior $P_\Phi(\boldsymbol{X}_p)$. For each such particle $\boldsymbol{x}_p$, we maintain some representation of the distribution $P_\Phi(\boldsymbol{X}_d \mid \boldsymbol{x}_p)$, and perform (exact or approximate) inference to compute the expectation of f relative to this distribution. For simplicity, we focus our discussion on Gibbs sampling; the extension to a Metropolis-Hastings algorithm with a general proposal distribution is straightforward.

We define the collapsed Gibbs sampling algorithm via a Markov chain whose states are instantiations to $\boldsymbol{X}_p$. As we discuss, to provide an unbiased estimator for the expectation over $P_\Phi(\boldsymbol{X}_p)$ in equation (12.29), we want the stationary distribution of this Markov chain to be $P_\Phi(\boldsymbol{X}_p)$. We thus modify our Gibbs sampling algorithm as follows: As in standard Gibbs sampling, we define a kernel for each variable $X_i \in \boldsymbol{X}_p$. Let $\boldsymbol{x}_{-i}$ be an assignment to $\boldsymbol{X}_p - \{X_i\}$. The kernel for X_i is defined as follows:

$$\mathcal{T}_i((\boldsymbol{x}_{-i}, x_i) \to (\boldsymbol{x}_{-i}, x_i')) = P_\Phi(x_i' \mid \boldsymbol{x}_{-i}). \tag{12.33}$$

This equation is very similar to equation (12.22). The only difference is that the event $\boldsymbol{x}_{-i}$ on which we condition the distribution over X_i is not a full assignment to all the other variables in the network, but rather only to the remaining variables in $\boldsymbol{X}_p$. Thus, the efficient computation of equation (12.23), where we simply compute the distribution over X_i given its Markov blanket, may not apply. However, we can compute this probability using inference in the network $\mathcal{H}_{\Phi[\boldsymbol{x}_p]}$ — the Markov network reduced over the assignment $\boldsymbol{x}_p$ — a model that we assume to be tractable. Indeed, the sampling approach can be well integrated with a clique tree over the variables $\boldsymbol{X}_p$ to allow the sampling process to be executed efficiently (see exercise 12.27).

Having defined the chain, we can use it in any of the ways described in section 12.3.5 to collect a data set $\mathcal{D}$ of particles $\boldsymbol{x}_p^{(m)}$, each of which is associated with a distribution over $\boldsymbol{X}_d$. Using these particles, we can estimate:

$$\hat{\boldsymbol{E}}_{\mathcal{D}}(f) = \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{E}_{P_{\Phi[\boldsymbol{x}_p[m]]}(\boldsymbol{X}_d)}[f(\boldsymbol{x}_p, \boldsymbol{x}_d, \boldsymbol{e})] \right).$$

Example 12.18

Consider the Markov network defined by the Bayesian network of example 12.11, reduced over the evidence $\boldsymbol{G} = \boldsymbol{g}$. This Markov network is a bipartite graph, where we have two sets of variables $\boldsymbol{I}, \boldsymbol{D}$ such that the only edges in the network are of the form I_j, D_k. Let $\phi_{j,k}(I_j, D_k)$ be the factor associated with each such edge. (For simplicity of notation, we assume that there is a factor for every such edge; the factors corresponding to grades that were not in the model will be vacuous.) We can now apply collapsed Gibbs sampling, selecting to sample the variables $\boldsymbol{D}$ and maintain a distribution over $\boldsymbol{I}$; this choice will (in most cases) be better, since there are usually more students than courses, and it is generally better to sample in a lower-dimensional space. Thus, our Markov

chain needs to be able to sample D_k from $P_\Phi(D_k \mid \boldsymbol{d}_{-k})$:

$$P_\Phi(d_k \mid \boldsymbol{d}_{-k}) = \frac{P_\Phi(d_1, d_2, \ldots, d_m)}{\sum_{D_k} P_\Phi(D_k, \boldsymbol{d}_{-k})}.$$

The expression in the numerator, and each term in the sum in the denominator, is the probability of a full assignment to $\boldsymbol{D}$. This type of expression can be computed as the partition function of the reduced Markov network that we obtain by setting $\boldsymbol{D} = \boldsymbol{d}$:

$$P_{\Phi[\boldsymbol{d}]}(I_1, \ldots, I_n) = \frac{1}{Z(\boldsymbol{d})} \prod_{j,k} \phi_{j,k}(I_j, d_k).$$

Each of the factors in the product is a singleton over an individual I_j. Thus, we can compute the partition function or marginals of this distribution in linear time. In particular, we can easily compute the Gibbs transition probability. Using the same analysis, we can easily compute an expression for $P_{\Phi[\boldsymbol{d}]}(\boldsymbol{I})$ in closed form, as a product of individual factors over the I_j's. Thus, we can also easily compute expectations of any function over these variables, such as the expected value of the number of smart students who got a grade of a C in an easy class (see exercise 12.28). ■

---

correspondence problem

data association

identity resolution

image registration

Box 12.D — Concept: Correspondence and Data Association. *One simple but important problem that arises in many settings is the* correspondence problem*: mapping between one set of objects $\mathcal{U} = \{u_1, \ldots, u_k\}$ and another $\mathcal{V} = \{v_1, \ldots, v_m\}$. This problem arises across multiple diverse application domains. Perhaps the most familiar are physical sensing applications, where $\mathcal{U}$ are sensor measurements and $\mathcal{V}$ objects that can generate such measurements; we want to know which object generated which measurement. For example, the objects may be airplanes, and the sensor measurements blips on a radar screen; or the objects may be obstacles in a robot's environments, and the sensor measurements readings from a laser range finder. (See box 15.A.) In this incarnation, this task is known as* data association*. However, there are also many other applications, of which we list only a few examples:*

- *Matching citations in text to the entities to which they refer (see, for example, box 6.D); this problem has been called* identity resolution *and* record matching.
- Image registration*, or matching features in an image representing one view of an object to features in an image representing a different view; this problem arises in applications such as* stereo reconstruction *or* structure from motion.
- *Matching words in a sentence in one language to words in the same sentence, translated to a different language; this problem is called* word alignment.
- *Matching genes in a DNA sequence of one organism to* orthologous genes *in the DNA sequence of another organism.*

This problem has been tackled using a range of models and a variety of inference methods. In this case study, we describe some of these approaches and the design decisions they made.

correspondence variable

Probabilistic Correspondence *To formulate the correspondence problem as a probabilistic model, we can introduce a set of* correspondence variables *that indicate the correspondence between one set of objects and the other. One approach is to have binary-valued variables C_{ij}, such that C_{ij} = true when u_i matches v_j, and false otherwise. While this approach is simple, it places no constraints on the number of matches for each i or for each j. Typically, we want to restrict our model so that, at least on one side, the match is unique. For example, in the radar tracking application, we typically want to assume that each measurement was derived from only a single object. In order to accommodate that in this model, we would need to add (hard)* mutual exclusion constraints *(*mutex*) that C_{ij} = true implies that $C_{ij'}$ = false for all $j' \neq j$. The resulting model is very densely connected, and it can be challenging for inference algorithms to deal with.*

mutual exclusion constraints

A more parsimonious representation uses nonbinary variables C_i such that $Val(C_i) = \{1, \ldots, m\}$; here, $C_i = j$ indicates that u_i is matched to v_j. The mutex constraints for the matches to u_i are then forced by the fact that each variable C_i takes on only a single value. Of course, we might also have mutex constraints in the other direction, where we want to assume that each v_j matches only a single u_i. We will return to this setting.

The probabilistic model and the evidence generally combine to produce a set of affinities w_{ij} that specify how likely u_i is to match to v_j. For convenience, we assume that these affinities are represented in log-space. These affinities can be derived from a generative probability — how likely is v_j to have generated u_i, or from an undirected potential that measures the quality of the match. (See, for example, box 6.D for an application where both approaches have been utilized.) For example, in the robot obstacle matching task, $\exp(w_{ij})$ may evaluate how likely it is that an obstacle v_j,at location L_j, generated a measurement u_i at sensed location S_i. In the citation matching problem, we may have a model that tells us how likely it is that an individual with the name "John X. Smyth" generated a citation "J. Smith." The affinities w_{ij} define a node potential *over the variables C_i: $\phi_i(C_i = j) = \exp(w_{ij})$. In addition, there may well be other components to the model, as we will discuss.*

The general inference task is then to compute the distribution over the correspondence variables or (as a fallback) to find the most likely assignment. We now discuss the challenges posed by this task in different variants of the correspondence problem, and some of the solutions.

The simplest case (which rarely arises in practice) is the one just described: We have only a set of node potentials $\phi_i(C_i = j)$ that specify the affinity of each u_i to each v_j. In this case, we have only a set of unrelated node potentials over the variables C_i, making the model one comprising independent random variables. We can then easily find the most likely assignment $c_i^* = \arg\max_j \phi_i(C_i = j)$, or even compute the full posterior $P(C_i) \propto \phi_i(C_i)$.

The inference task becomes much more challenging when we extend the model in a way that induces correlations over the correspondence variables. We now describe three settings where such complications arise, noting that these are largely orthogonal, so that more than one of these complicating factors may arise in any given application.

Two-Sided Mutex Constraints *The first complication arises if we want to impose mutex constraints not just on the correspondence of the u_i's to the v_j's, but also in the reverse direction; that is, we want each v_j to be assigned to exactly one u_i. (We note that, unless $k = m$, a perfect match is not possible; however, we can circumvent this detail by adding "dummy" objects that are equally good matches to anything.) This requirement induces a model where all of the C_i's are connected to each other with potentials that impose mutex constraints, making it clearly intractable for a*

variable elimination algorithm. Nevertheless, several techniques have been effectively applied to this problem.

bipartite matching

MAP assignment

First, we note that we can view the correspondence problem, in this case, as a bipartite graph, where the u_i's are one set of vertices, the v_j's are a second set, and an edge of weight w_{ij} connects u_i to each v_j. An assignment satisfying the mutex constraints is a bipartite matching *in this graph — a subset of edges such that each node has exactly one adjacent edge in the matching. Finding the single highest-probability assignment — the* MAP assignment *— is equivalent to one of finding the bipartite matching whose weight (sum of the edge weights in the matching) is largest. This problem can be solved very efficiently using combinatorial optimization algorithms: If we can match any u_i to any v_j, the total running time is $O(k^3)$; if we have only ℓ possible edges, the running time is $O(k^2 \log(k) + k\ell)$.*

Of course, in many cases, a single assignment is not an adequate solution for our problem, since there may be many distinct solutions that have high probability. For computing posteriors in this setting, two main approaches have been used.

The first is an MCMC sampler over the space of possible matchings. Here, Gibbs sampling is not a viable approach, since any change to the value of a single variable would give rise to an assignment that violates the mutex constraints. A Metropolis-Hastings chain is a good solution, but for good performance, the proposal distribution must be carefully chosen. Most obvious is to pick two variables such that $C_{i_1} = j_1$ and $C_{i_2} = j_2$, and propose a move that flips them, setting $C_{i_1} = j_2$ and $C_{i_2} = j_1$. While this approach is a legal chain, it can be slow to mix: in most local maxima, a single flip of this type will often produce a very poor solution. Thus, some work has been done on constructing proposal distributions with larger steps that flip multiple variables at a time while still maintaining a legal matching.

The second solution uses loopy belief propagation over the Markov network with the C_i variables. As we mentioned, however, the mutex constraints give rise to a fully connected network, which is a very challenging setup for belief propagation. Here, however, we can adopt a simple heuristic: We begin without including the mutex constraints and run inference. We then examine the resulting posterior and identify those mutex constraints that are violated with high probability (those where $P(C_{i_1} = C_{i_2})$ is high). We then add those violated constraints and repeat the inference. In practice, this process usually converges long before all k^2 mutex constraints are added.

Unobserved Attributes *The second type of complication arises when the affinities w_{ij} depend on properties $\boldsymbol{A}_j$ of the objects that are unobserved, and need to be inferred. For example, in the citation matching problem, we generally do not know the correct name of an author or a paper; given the name of an author v_j, we can determine the affinity w_{ij} of any citation u_i. Conversely, given a set of citations u_i that match v_j, we can infer the most likely name to have generated the observed names in these citations. The same situation arises in many other applications. For example, in the airplane tracking application, the plane's location at a given point in time is generally unknown, although we may have a prior on this location based on observations at previous time points. Given the location of airplane v_j, we can determine how likely it was to have generate the blip u_i. Conversely, once we assign blip u_i to v_j, we can update our posterior on v_j's location.*

More formally, here we have a set of observed features $\boldsymbol{B}_i$ for each object u_i, and a set of hidden attributes $\boldsymbol{A}_j$ for each v_j. We have a prior $P(\boldsymbol{A}_j)$, and a set of factors $\phi_i(\boldsymbol{A}_j, \boldsymbol{B}_i, C_i)$ that are vacuous (uniform) if $C_i \neq j$. We want to compute the posterior over $\boldsymbol{A}_j$. For this problem,

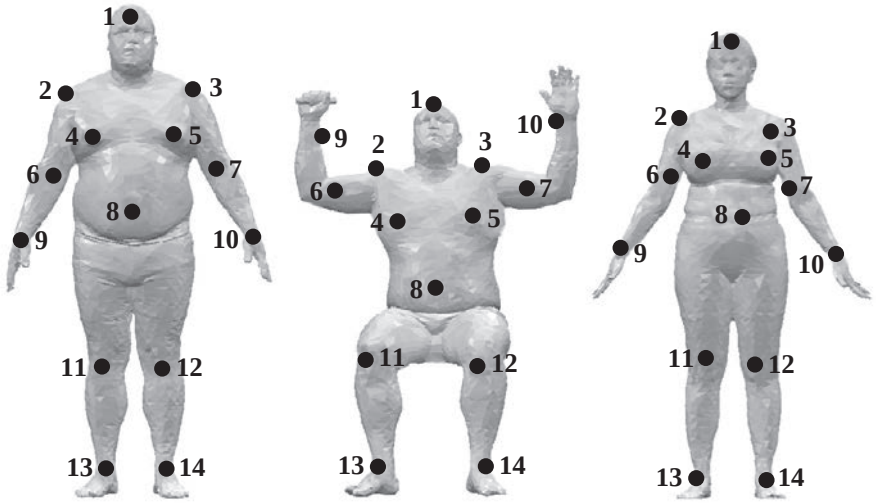


Figure 12.D.1 — Results of a correspondence algorithm for 3D human body scans The model explicitly captures correlations between different correspondence variables.

one obvious extension of the MCMC approach we described is to use a collapsed MCMC approach, where we sample the correspondence variables but maintain a closed-form distribution over $\boldsymbol{A}_j$; see exercise 12.29. When the features $\boldsymbol{A}_k$ are such that maintaining a closed-form posterior is challenging (for example, they are continuous and do not have a simple parametric form), we often adopt an approach where we pick a single assignment to each $\boldsymbol{A}_j$; this solution can be implemented in the framework of an EM *algorithm (see box 19.D for one example).*

EM

Directly Correlated Correspondences *A final complication arises when we wish to model direct* correlations *between the values of different correspondence variables. This type of correlation would arise naturally, for example, in an application where we match visual features (small patches) in two images of the same real-world object (say a person). Here, we not only want the local appearance of a feature to be reasonably similar in the two images; we also want the relative geometry of the features to be consistent. For example, if a patch containing the eye is next to a patch containing the nose in one image, they should also be close in the other. These (soft) constraints can be represented as potentials on pairs of variables C_{i_1}, C_{i_2} (or even on larger subsets). Similar situations arise in sequence alignment, where we often prefer to match adjacent sequence elements (whether words or genes) on one side to adjacent sequence elements on the other.*

correlated correspondence

Here, we can often exploit the fact that the spatial structure we are trying to encode is local, so that the resulting Markov network over the C_i variables is often reasonably structured. As a consequence, techniques such as loopy belief propagation, if carefully designed, can be surprisingly effective in this setting.

Figure 12.D.1 demonstrates the results of one model along these lines. The model, due to Anguelov et al. (2004), aims to find a correspondence between a set of (automatically selected) landmarks

on different three-dimensional scans of human bodies. Here, $\phi_i(C_i = j)$ represents the extent to which the local appearance around the ith landmark in one scan is similar to the local appearance of the jth landmark on the other. This task is very challenging for two reasons. First, the local appearance of corresponding patches on two scans can be very different, so that the node potentials are only somewhat informative. Second, the two scans exhibit significant deformation, both of shape and of pose, so that standard parametric models of deformation do not work. In this task, modeling correlations between the different correspondence variables allows us to capture constraints regarding the preservation of the object geometry, specifically, the fact that distances between corresponding points should be roughly preserved. This feature was essential for obtaining reasonable correspondences. Most of the constraints regarding preservation of distances relate to pairs of nearby points, so that the resulting Markov network was not too densely connected, allowing a judicious application of loopy belief propagation to work well.

12.5 Deterministic Search Methods ⋆

So far, we have focused on particles generated by random sampling. Random sampling tries to explore the state space of the distribution "uniformly," generating each state proportionately to its probability. **A sampling-based approach can, however, be problematic when we have a highly skewed distribution, where only a small number of states have nonnegligible probability. In this case, sampling methods will tend to sample the same small set of states repeatedly, wasting computational resources to no gain. An alternative approach, designed for settings such as this, is to use a deterministic method that explicitly searches for high-probability states.**

Intuitively, in these search methods, we deterministically generate some set of distinct assignments $\mathcal{D} = \{\xi[1], \ldots, \xi[M]\}$. We then approximate the joint distribution P by considering only these instantiations, ignoring the rest. Intuitively, if our particles account for a large proportion of the probability mass, we have a reasonable approximation to the joint.

Example 12.19 *In the* Student *network of figure 12.1, the most likely ten instantiations (of 48), in decreasing order, are:*

d^1	i^0	g^3	s^0	l^0	0.184
d^0	i^0	g^3	s^0	l^0	0.119
d^0	i^1	g^1	s^1	l^1	0.117
d^0	i^0	g^1	s^0	l^1	0.108
d^0	i^0	g^2	s^0	l^1	0.096
d^0	i^0	g^2	s^0	l^0	0.064
d^1	i^1	g^1	s^1	l^1	0.043
d^1	i^0	g^2	s^0	l^1	0.04
d^0	i^1	g^1	s^0	l^1	0.029
d^1	i^0	g^2	s^0	l^0	0.027.

Together, they account for about 82.6 percent of the probability mass. To account for 95 percent of the probability mass, we would need twenty-two instantiations, and to account for 99 percent of the mass, we would need thirty-three instantiations. ■

Intuitively, the quality of our particle-based approximation to the joint distribution improves with the amount of probability mass accounted for in our particles. Thus, our goal is to enumerate instantiations that have high probability in some unnormalized measure $\tilde{P}$. Specifically, in the case of Bayesian networks, we want to enumerate instantiations that are likely given $\boldsymbol{e}$.

We can formalize this goal as one of finding the highest-probability instantiations in $\tilde{P}$, until we reach some upper bound K on the number of particles. (One might be tempted to use, as a stopping criterion, a lower bound on the total mass accumulated in the enumerated particles; however, because we generally do not know the normalizing constant, the absolute mass accumulated has no real interpretation.) Clearly, this problem encompasses within it the task of finding the single most likely instantiation, also known as the *maximum a posteriori (MAP) assignment*. The problem of finding the MAP assignment is the focus of chapter 13. Not surprisingly, many of the successful methods for enumerating high-probability assignments are extensions of methods for finding the MAP assignment. Thus, we largely defer discussion of methods for finding high-probability particles to that chapter. (See, for example, exercise 15.10.) In this section, we focus primarily on the question of using a set of deterministically selected particles to approximate the answers to conditional probability queries.

MAP assignment

Consider any event $\boldsymbol{Z} = \boldsymbol{z}$. Approximating the joint using the set of high-probability assignments $\mathcal{D}$, we have that one natural estimate for $\tilde{P}(\boldsymbol{z})$ is:

$$\sum_{m=1}^{M} \mathbf{1}\{\boldsymbol{z}[m] = \boldsymbol{z}\}\tilde{P}(\xi[m]), \tag{12.34}$$

where $\boldsymbol{z}[m] = \xi[m]\langle \boldsymbol{Z}\rangle = \boldsymbol{z}$. It is important to note a key difference between this estimate and the one of equation (12.2). There, we merely counted particles, whereas in this case, the particles are weighted by their probability. Intuitively, the difference is due to the fact that, in sampling methods, particles are generated proportionately to their probability. Thus, higher- probability instantiations are generated more often. If we then also weighted the sampled particles by their probability, we would be "double-counting" the probability. Alternatively, we can view this formula as an instance of the importance sampling estimator, equation (12.8). In this case, each particle $\xi[m]$ is generated from a (different) deterministic proposal distribution Q, that ascribes it probability 1.[4] Hence, we must weight the particle by $\tilde{P}(\xi[m])/Q(\xi[m]) = \tilde{P}(\xi[m])$.

Here, however, we have no guarantees about the bias of the estimator. Depending on our search procedure, we might end up with estimates that are arbitrarily bad. At one extreme, for example, we might have a search procedure that avoids (for as many particles as possible) any instantiation ξ where $\xi\langle \boldsymbol{Z}\rangle = \boldsymbol{z}$. Clearly, our estimate will then be biased in favor of low values. A more correct approach is to use our particles to provide both an upper and lower bound to the unnormalized probability of any event $\boldsymbol{z}$. Some of our particles $\xi[m]$ have $\boldsymbol{z}[m] = \boldsymbol{z}$. The total probability mass of these particles is a *lower bound* on the probability mass of all instantiations ξ where $\boldsymbol{Z} = \boldsymbol{z}$. Similarly, the total probability mass of particles that have $\boldsymbol{z}[m] \neq \boldsymbol{z}$ is a lower bound on the complementary mass, and hence provides an upper bound on the probability

lower bound

4. We cannot, of course, actually apply importance sampling in this way, since this deterministic proposal distribution violates our assumption that Q's support contains that of P. However, this perspective provides intuition for our choice of weights.

mass of the assignments where $\boldsymbol{Z} = \boldsymbol{z}$:

$$\sum_{m=1}^{M} \boldsymbol{I}\{\boldsymbol{z}[m] = \boldsymbol{z}\}\tilde{P}(\xi[m]) \leq \tilde{P}(\boldsymbol{Z} = \boldsymbol{z}) \leq \left(1 - \sum_{m=1}^{M} \boldsymbol{I}\{\boldsymbol{z}[m] \neq \boldsymbol{z}\}\tilde{P}(\xi[m])\right). \tag{12.35}$$

Equivalently, we can define $\rho = 1 - \sum_{m=1}^{M} \tilde{P}(\xi[m])$ to be the probability mass not accounted for by our particles. The bound can then be rewritten as:

$$\sum_{m=1}^{M} \boldsymbol{I}\{\boldsymbol{z}[m] = \boldsymbol{z}\}\tilde{P}(\xi[m]) \leq \tilde{P}(\boldsymbol{Z} = \boldsymbol{z}) \leq \rho + \sum_{m=1}^{M} \boldsymbol{I}\{\boldsymbol{z}[m] = \boldsymbol{z}\}\tilde{P}(\xi[m]).$$

This reformulation reflects the fact that the unaccounted probability mass can be associated either with the event $\boldsymbol{Z} = \boldsymbol{z}$ or with its complement. If all of the unaccounted probability mass is associated with $\boldsymbol{z}$, we get the upper bound, and if all of it is associated with the complement, we get the lower bound.

Example 12.20

Consider the Student *network of figure 12.1, and the event l^1 — a strong letter. Within the ten particles of example 12.19, the particles compatible with this event are: $\xi[3]$, $\xi[4]$, $\xi[5]$, $\xi[7]$, $\xi[8]$, and $\xi[9]$. Together, they account for 0.433 of the probability mass in the joint. The complementary event accounts for 0.393 of the mass. The total unaccounted mass is $1 - 0.826 = 0.174$. From these numbers, we obtain the following bounds:*

$$0.433 \leq P(l^1) \leq 1 - 0.393 = 0.433 + 0.174 = 0.607.$$

The true probability of this event is 0.502.

Now, consider the event i^0, l^1 — a weak student getting a strong letter. In this case, we have three instantiations compatible with this assignment: $\xi[4]$, $\xi[5]$, and $\xi[8]$. Together, they account for 0.244 of the probability mass. The remaining assignments, which are incompatible with the event, account for 0.582 of the mass. Altogether, we obtain the following bounds for the probability of this event:

$$0.244 \leq P(i^0, l^1) \leq (1 - 0.582) = 0.418.$$

The true probability of this event is 0.272.

When evaluating these results, it is important to note that they are based on the use of only ten particles. The results from any sampling-based method that uses only ten particles would probably be significantly worse. ■

We can see that the true probability can lie anywhere within the interval specified by equation (12.35), and there is no particular justification for choosing any particular point within the range (for example, the point specified by equation (12.34)). If our search happens to first find instantiations that are compatible with $\boldsymbol{z}$, the lower bound is likely to be a better estimate; if it first finds instantiations that are incompatible with $\boldsymbol{z}$, the upper bound is likely to be closer.

The fact that we obtain both lower and upper bounds on the mass in the unnormalized measure also allows us to provide bounds on the value of probability (or conditional) probability queries. Assume we are trying to compute $P(\boldsymbol{y} \mid \boldsymbol{e}) = \tilde{P}(\boldsymbol{y}, \boldsymbol{e})/\tilde{P}(\boldsymbol{e})$. We can now obtain upper and lower bounds for both the numerator and denominator, using equation (12.35). A lower

bound for the numerator and an upper bound for the denominator provide a lower bound for the ratio. Similarly, an upper bound for the numerator and a lower bound for the denominator provide an upper bound for the ratio. Analogously, we can obtain bounds on the marginal probability $P(\boldsymbol{y})$ by normalizing $\tilde{P}(\boldsymbol{y})$ by $\tilde{P}(\mathit{true})$.

More precisely, assume that we have bounds:

$$\begin{array}{ccccc} \ell_{\boldsymbol{y},\boldsymbol{e}} & \leq & P(\boldsymbol{y},\boldsymbol{e}) & \leq & u_{\boldsymbol{y},\boldsymbol{e}} \\ \ell_{\boldsymbol{e}} & \leq & P(\boldsymbol{e}) & \leq & u_{\boldsymbol{e}}. \end{array}$$

Then we can bound:

$$\frac{\ell_{\boldsymbol{y},\boldsymbol{e}}}{u_{\boldsymbol{e}}} \leq P(\boldsymbol{y} \mid \boldsymbol{e}) \leq \frac{u_{\boldsymbol{y},\boldsymbol{e}}}{\ell_{\boldsymbol{e}}}. \tag{12.36}$$

Example 12.21

Assume we want to compute $P(i^0 \mid l^1)$ — the probability that a student with a strong letter is not intelligent — using our ten samples from example 12.19. We have already shown the bounds for both the numerator and the denominator. We obtain:

$$\begin{array}{ccccc} 0.265/0.607 & \leq & P(i^0 \mid l^1) & \leq & 0.439/0.433 \\ 0.437 & \leq & P(i^0 \mid l^1) & \leq & 1.014. \end{array}$$

The true probability of this event is 0.542, which is fairly far away from any of the bounds. ■

Interestingly, the upper bound in this case is greater than 1. Although clearly valid, this conclusion is not particularly interesting. In general, the deterministic approximations are of value only if we can cover a very large fraction of our probability mass with a small number of particles. While this constraint might seem very restrictive (as it often is), there are nevertheless applications where the probabilities are very skewed, and this assumption is a very good one; we return to this point in section 12.6.

A very similar discussion applies to the extension of deterministic search methods to collapsed particles. In this case, we approximate the outer expectation in equation (12.29) using a set $\mathcal{D}$ of particles $\boldsymbol{x}_p[1], \ldots, \boldsymbol{x}_p[M]$, which are selected using some deterministic search procedure. As usual, each particle is associated with a distribution $P(\boldsymbol{X}_d \mid \boldsymbol{x}_p[m], \boldsymbol{e})$.

Consider the task of computing the probability $P(\boldsymbol{z})$. As in the case of full particles, each of the generated particles $\boldsymbol{x}_p[m]$ accounts for a certain part of the probability mass. However, in this case, we cannot simply test whether the event $\boldsymbol{z}$ is compatible with the particle, since the particle might not specify a full assignment to the variables $\boldsymbol{Z}$. Rather, we have to compute the probability of $\boldsymbol{z}$ relative to the distribution over all of the variables defined by the particle. (See exercise 12.26.)

Thus, in particular, the lower bound on $P(\boldsymbol{z})$ defined by our particles is:

$$\begin{aligned} &\sum_{m=1}^{M} P(\boldsymbol{x}_p[m]) \left(\boldsymbol{E}_{P(\boldsymbol{X}_d \mid \boldsymbol{x}_p[m])}[\boldsymbol{1}\{\boldsymbol{x}_p, \boldsymbol{X}_d\langle \boldsymbol{Z} \rangle = \boldsymbol{z}\}] \right) \\ &\quad = \sum_{m=1}^{M} P(\boldsymbol{x}_p[m]) P(\boldsymbol{z} \mid \boldsymbol{x}_p[m]). \end{aligned}$$

The lower bound assumes that none of the unaccounted probability mass is compatible with $\boldsymbol{z}$. Similarly, the upper bound assumes that all of this unaccounted mass is compatible with $\boldsymbol{z}$, leading to:

$$\left(\sum_{m=1}^{M} P(\boldsymbol{z}, \boldsymbol{x}_p[m])\right) + \left(1 - \sum_{m=1}^{M} P(\boldsymbol{x}_p[m])\right).$$

Once again, we can compute both the probability $P(\boldsymbol{z} \mid \boldsymbol{x}_p)$ and the weight $P(\boldsymbol{x}_p)$ by using inference in the conditioned network.

incremental conditioning

This method is simply an incremental version of the *conditioning* algorithm of section 9.5, using $\boldsymbol{x}_p$ as a conditioning set. However, rather than enumerating all possible assignments to the conditioning set, we enumerate only a subset of them, attempting to cover most of the probability mass. This algorithm is also called *bounded conditioning*.

bounded conditioning

12.6 Summary

This chapter presents a series of methods that attempt to approximate a joint distribution using a set of particles. We discussed three main classes of techniques for generating particles.

Importance sampling, and specifically likelihood weighting, generates particles by random sampling from a distribution. Because we cannot, in general, sample from the posterior distribution, we generate particles from a different distribution, called the proposal distribution, and then adjust their weights so as to get an unbiased estimator. The proposal distribution is a mutilated Bayesian network, and the samples are generated using forward sampling. Owing to the use of forward sampling, likelihood weighting applies only to directed graphical models. However, if we somehow choose a proposal distribution, the more general framework of importance sampling can also be applied to undirected graphical models (see exercise 12.9).

Markov chain Monte Carlo techniques attempt to generate samples from the posterior distribution. We define a Markov chain, or a stochastic sampling process, whose stationary distribution (the asymptotic result of the sampling process) is the correct posterior distribution. The Metropolis-Hastings algorithm is a general scheme for specifying a Markov chain that induces a particular posterior distribution. We showed how we can use the Metropolis-Hastings algorithm for graphical models. We also showed a particular instantiation, called Gibbs sampling, that is specifically designed for graphical models.

Finally, search methods take a different approach, where particles are generated deterministically, trying to focus on instantiations that have high probability. Unlike random sampling methods, deterministic search methods do not provide an unbiased estimator for the target query. However, they do provide sound upper and lower bounds. When the probability distribution is highly diffuse, so that many particles are necessary to cover most of the probability mass, these bounds will be very loose, and generally without value. However, when a small number of instantiations account for a large fraction of the probability mass, deterministic search techniques can obtain very accurate results with a very small number of particles, often much more accurate results than sampling-based methods with a comparable number of particles. There are several applications that have this property. For example, when performing fault diagnosis (see, for example, box 5.A), where faults are very rare, it can be very efficient to enumerate all hypotheses where the system has up to K faults. Because multiple faults are highly unlikely,

even a small value of K (2 or 3) will likely suffice to cover most of the probability mass — even the mass consistent with our evidence. Another example is speech recognition (box 6.B), where only very few trajectories through the HMM are likely. In both of these applications, deterministic search methods have been successfully applied.

From a high level, it appears that sampling methods are the ultimate general-purpose inference algorithm. They are the only method that can be applied to arbitrary probabilistic models and that is guaranteed to achieve the correct results at the large sample limit. Indeed, when faced with a complex probabilistic model that involves continuous variables or a nonparametric model, there are often very few other choices available to us. While optimization-based methods, such as those of chapter 11, can sometimes be applied, the application often requires a nontrivial derivation, specific to the problem at hand. Moreover, these methods provide no accuracy guarantee. Conversely, it seems that sampling-based methods can be applied easily, off-the-shelf, to virtually any model.

This impression, however, is somewhat misleading. **While it is true that sampling methods provide asymptotic guarantees, their performance for reasonable sample sizes is very difficult to predict. In practice, a naive application of sampling methods to a complex probabilistic model often fails dismally, in that the estimates obtained from any reasonable number of samples are highly inaccurate. Thus, the success of these methods depends heavily on the properties of the distribution, and on a careful design of our sampling algorithm. Moreover, there is little theoretic basis for this design, so that the process of getting sampling methods to work is largely a matter of intuition and intensive experimentation.**

Nevertheless, the methods described in this chapter do provide an important component in our arsenal of methods for inference in complex models. Moreover, they are often used very successfully in combination with exact or approximate global inference methods. Standard combinations include the use of global inference for providing more informed proposal distributions, and for manipulating collapsed particles. Such combinations are highly successful in practice, and they often lead to much better results than any of the two types of inference methods in isolation.

Having described these basic methods, we showed how they can be extended to the case of collapsed particles, which consist of an assignment to a subset of network variables, associated with a closed-form distribution over the remaining ones. The answer to a query is then a (possibly weighted) sum over the particles, of the answer to the query within each associated distribution. This approach approximates part of the inference task via particles, while performing exact inference on a subnetwork, which may be simpler than the original network.

12.7 Relevant Literature

The $\mathcal{NP}$-hardness of approximate probabilistic inference in Bayesian networks was shown by Dagum and Luby (1993).

There is a vast literature on the use of Monte Carlo methods for estimating integrals in general, and the expectation of functions in particular. See Robert and Casella (2005) for one good review. Geweke (1989) proves some of the basic results regarding the accuracy of the importance sampling estimates.

Probabilistic logic sampling was first proposed by Henrion (1986). The improvement to likelihood weighting was proposed independently by Fung and Chang (1989) and by Shachter and Peot (1989). Cano et al. (2006) and Dagum and Luby (1997) proposed a variant of likelihood weighting based on unnormalized importance sampling, which separately estimates $P(\boldsymbol{y}, \boldsymbol{e})$ and $P(\boldsymbol{e})$, as described in section 12.2.3.2. Dagum and Luby also proposed the use of a data-dependent stopping rule for the case where the CPD entries are bounded away from 0 or 1. For this case, they provide guaranteed bounds on the expected number of samples required to achieve a certain error rate, as discussed in section 12.2.3.2. Pradhan and Dagum (1996) provide some empirical validation of this algorithm, applied to a large medical diagnosis network.

Various heuristic approaches for improving the proposal distribution have been proposed. Fung and del Favero (1994) proposed the backward sampling algorithm, that allows for generating samples from evidence nodes in the direction that is opposite to the topological order of nodes in the network, combining their likelihood function with the CPDs of some previously sampled variables. Other variants use some alternative form of approximate inference over the network to produce an approximation $Q(\boldsymbol{X})$ to $P(\boldsymbol{X} \mid \boldsymbol{E} = \boldsymbol{e})$, and then use Q as a proposal distribution for importance sampling. For example, de Freitas et al. (2001) use variational methods (described in chapter 11) in this way.

Shachter and Peot (1989) proposed an adaptive approach, called *self-importance sampling*, which adapts the proposal distribution to the samples obtained, attempting to increase the probability of sampling in higher-probability regions of the posterior distribution $P(\boldsymbol{X} \mid \boldsymbol{e})$. This approach was subsequently improved by Cheng and Druzdzel (2000) and by Ortiz and Kaelbling (2000), who proposed an *adaptive importance sampling* method that uses the variance of the estimator in a more principled way.

adaptive importance sampling

Shwe and Cooper (1991) applied importance sampling to the QMR-DT network for medical diagnosis (Shwe et al. 1991). Their variant of the algorithm combined self-importance sampling and an improved proposal distribution called Markov blanket scoring. This proposal distribution was designed to be computed efficiently in the context of BN2O networks.

Sampling methods based on Markov chains were first proposed for models arising in statistical physics. In particular, the Metropolis-Hastings algorithm was first proposed by Metropolis et al. (1953). Geman and Geman (1984) applied Gibbs sampling to image restoration in a paper that was very influential in popularizing this method within computer vision, and subsequently in related communities. Ackley, Hinton, and Sejnowski (1985) propose the use of Gibbs sampling within Boltzmann machines, for both inference and parameter estimation. Pearl (1987) introduced Gibbs sampling for Bayesian networks. York (1992) continues this work, specifically addressing the problem of networks where some states have low (or even zero) probability.

Over the years, extensive work has been done on the topic of MCMC methods, addressing a broad range of topics including: theoretical analyses, application to new classes of models, improved algorithms that are faster or have better convergence properties, specific applications, and many more. Works reviewing some of these developments include Neal (1993); Smith and Roberts (1993); Gilks et al. (1996); Gamerman and Lopes (2006). Neal (1993), in particular, provides both an excellent tutorial, guidelines for practitioners, and a comprehensive annotated bibliography for relevant papers (up to 1993). Tierney (1994) discusses the conditions under which we can use multiple kernels within a single chain. Nummelin (1984, 2002) shows the central limit theorem for samples from a Markov chain. MacEachern and Berliner (1994) show that subsampling the samples derived from a Markov chain is suboptimal. Gelman and Rubin

(1992) provide a specific strategy for applying MCMC: they specify the number of chains, the burn-in time, and the intervals at which samples should be selected. Their strategy is applicable mostly for problems for which a good initial distribution is available, but provides insights more broadly.

Algorithms that improve convergence are particularly relevant for the high-dimensional, multimodal distributions that often arise in the setting of graphical models. Some methods for addressing this issue use larger, nonlocal steps in the search space, which are helpful in breaking out of local optima; for example, for pairwise MRFs where all variables have a uniform set of values, Swendsen and Wang (1987) and Barbu and Zhu (2005) propose moves that simultaneously flip an entire subgraph from one value to another. Higdon (1998) discusses the general idea of introducing auxiliary variables as a mechanism for taking larger steps in the space. The temperature-based methods draw on the idea of simulated annealing (Kirkpatrick et al. 1983). These methods include simulated tempering (Marinari and Parisi 1992; Geyer and Thompson 1995) in which the state of the model is augmented with a temperature variable for purposes of sampling; parallel tempering (Swendsen and Wang 1986; Geyer 1991) runs multiple chains at different temperatures at the same time and allows chains to exchange datapoints; tempered transitions (Neal 1996) proposes a new sample by moving up and down the temperature schedules; and *annealed importance sampling* Neal (2001) uses a similar approach in combination with an importance sampling reweighting scheme.

annealed importance sampling

The BUGS system is an invaluable tool for the application of MCMC methods, inasmuch as it encompasses a large class of models and implements a number of fairly advanced techniques for improving the mixing rate. The system itself, and some of the ideas in it, are described by Thomas et al. (1992) and Gilks et al. (1994).

The task of finding high-probability assignments in a probabilistic model is very closely related to the problem of finding a MAP assignment. We refer the reader to section 13.9 for many of the relevant references on that topic.

Techniques based on deterministic search were popular in the early days of the Bayesian network community, often motivated by connections with search algorithms and constraint-satisfaction algorithms in AI. As a few examples, Henrion (1991) proposes a search-based algorithm aimed specifically at BN2O networks, whereas Poole (1993b, 1989) describes an algorithm aimed at more general network structures. Horvitz et al. (1989) propose an algorithm that combines conditioning with search, to obtain some of the benefits of exact inference in a search-based algorithm. More recently, Lerner et al. (2000) use a collapsed search-based approach for the problem of fault diagnosis, where probabilities are also highly skewed.

The use of collapsed particles, and specifically the Rao-Blackwell theorem, for sampling-based estimation was proposed by Gelfand and Smith (1990) and further developed by Liu et al. (1994) and Robert and Casella (1996). This idea has since been used in many applications of sampling techniques to graphical models, where sampling some of the variables can often allow us to perform tractable inference over the others. The idea of sampling some of the variables in a Bayesian network and performing exact inference over the others was explored by Bidyuk and Dechter (2007).

We have focused our presentation here on the problem of generating samples from a distribution so as to estimate the posterior. However, another task of significant practical importance is that of computing the *partition function* of an unnormalized measure. As we will see, this task is of particular importance in learning, since the normalizing constant is often used as

partition function

a measure of the overall quality of a learned model. Computation of a partition function by directly sampling the distribution leads to estimates of high variance, since such estimates are usually dominated by a small number of samples with high unnormalized probabilities. In order to avoid this problem, a ratio of partition functions is computed; see exercise 12.8. An equivalent problem of free energy difference, or partition function ratio, has been tackled in computational physics and computation chemistry communities with a range of methods, including free energy perturbation, thermodynamic integration, bridge sampling, umbrella sampling, and Jarzynski equation; these methods are in essence importance-sampling algorithms. See Gelman and Meng (1998); Jarzynski (1997); Neal (2001) for some examples.

correspondence

Extensive work has been done on the *correspondence* problem in its various incarnations: data association, record matching, identity resolution, and more. See, for example, Bar-Shalom and Fortmann (1988); Bar-Shalom (1992) for a review of some of the key ideas. Pasula et al. (1999) first proposed the use of MCMC methods to sample the space of possible associations in target tracking, based on the analysis of Jerrum and Sinclair (1997) for sampling over matchings. Dellaert et al. (2003) also use MCMC for the data-association problem in a structure-from-motion task in computer vision; they propose a more sophisticated proposal distribution that allows more rapid mixing. Anguelov et al. (2004) suggest the use of belief propagation for solving the correspondence problem, in settings where the correspondences are correlated with each other. Their results form part of box 12.D. Some work has also been done on finding the MAP assignment to a data-association problem using variants of belief propagation (Chen et al. 2003; Duchi et al. 2006) or other methods (Lacoste-Julien et al. 2006).

12.8 Exercises

Exercise 12.1⋆

Consider the sequence of variables T_n defined in appendix A.2. Given ϵ and δ, we define $m(\epsilon, \delta) = \min_n P(|T_n - p| \geq \epsilon) \leq \delta$ to be smallest number of trials we need to ensure that the probability of deviating ϵ standard deviations from p is less than δ. Although we cannot compute $m(\epsilon, \delta)$ exactly, we can upper-bound it using the bounds we discuss earlier.

a. Use Chebyshev's bound to give an upper bound on $m(\epsilon, \delta)$. (You can use the fact that $\mathbb{V}ar[T_n] \leq \frac{1}{4n}$.)
b. Use the Chernoff bounds discussed above to give an upper bound on $m(\epsilon, \delta)$.
c. What is the difference between these two bounds? How does each of these depend on ϵ and δ? What are the implications if we want design a test to estimate p?

Exercise 12.2

Consider the problem of providing a reliable estimate for a ratio p/q, where we have estimators $\hat{p}$ for p and $\hat{q}$ for q.

a. Assume that $\hat{p}$ has ϵ relative error to p and $\hat{q}$ has ϵ relative error to q. More precisely, $\hat{p} \in [(1 - \epsilon)p, (1 + \epsilon)p]$, and $\hat{q} \in [(1 - \epsilon)q, (1 + \epsilon)q]$. Provide a relative error bound on $\hat{p}/\hat{q}$ relative to p/q.
b. Now, assume that we have only an absolute error bound for $\hat{p}$ and $\hat{q}$: $\hat{p} \in [p-\epsilon, p+\epsilon]$, $\hat{q} \in [q-\epsilon, q+\epsilon]$. Show by example that $\hat{p}/\hat{q}$ can be an arbitrarily bad estimate for p/q.
c. What type of guarantee (if any) can you provide if $\hat{p}$ has low absolute error but $\hat{q}$ has low relative error?

Exercise 12.3

Assume we have a calibrated clique tree for a distribution P_Φ. Show how we can use the clique tree to generate samples that are sampled exactly from P_Φ.

Exercise 12.4

Prove proposition 12.2.

Exercise 12.5⋆

Let $\mathcal{B}$ be a Bayesian network, and X a variable that is not a root of the network. Show that $P_{\mathcal{B}}$ may not be the optimal proposal distribution for estimating $P_{\mathcal{B}}(X = x)$.

Exercise 12.6

edge reversal

In exercise 3.12, we defined the *edge reversal* transformation, which reverses the directionality of an edge $X \to Y$. How would you apply such a transformation in the context of likelihood weighting, and what would be the benefits?

Exercise 12.7⋆

Let $\mathcal{B}$ be a network, $\boldsymbol{E} = \boldsymbol{e}$ an observation, and $X_1, \ldots, X_k$ be some ordering (not necessarily topological) of the unobserved variables in $\mathcal{B}$. Consider a sampling algorithm that, for each $X_1, \ldots, X_k$ in order, randomly selects a value $x_i[m]$ for X_i using some distribution $Q(X_i \mid x_1[m], \ldots, x_{i-1}[m], \boldsymbol{e})$.

a. Write the formula for the importance weights in normalized importance sampling using this sampling process.
b. Using your answer in part 1, define an improved likelihood weighting algorithm that samples variables in the network in topological order, but, for a variable X_i with parents $\boldsymbol{U}_i$, samples X_i using a distribution that uses both $\boldsymbol{x}_{-i}[m]$ and the evidence in X_i's Markov blanket.
c. Does your approach from part 2 generate samples from the posterior distribution $P(X_1, \ldots, X_k \mid \boldsymbol{e})$? Explain.

Exercise 12.8⋆

Consider the normalized importance sampling algorithm, but now assume that, in equation (12.10), an unnormalized measure $\tilde{Q}(\boldsymbol{X})$ is used in place of $Q(\boldsymbol{X})$. Show that the average of the weights converges to $\frac{\sum_{\boldsymbol{X}} \tilde{P}(\boldsymbol{x})}{\sum_{\boldsymbol{X}} \tilde{Q}(\boldsymbol{X})}$, which is the ratio of the normalizing constants of the two distributions.

Exercise 12.9

In this question, we consider the application of importance sampling to Markov networks.

a. Explain intuitively why we cannot simply apply likelihood weighting to Markov networks.
b. Show how likelihood weighting can be applied to chordal Markov networks. Is this approach interesting? Explain.
c. Provide a technique by which the more general framework of importance sampling can be applied to Markov networks. Be sure to define both a reasonable proposal distribution and an algorithmic technique for computing the weights.

Exercise 12.10⋆

Consider the Grasshopper example of figure 12.3

a. Assume that the probability space is the full set of (positive and negative) integers; the transition model is now the same for all i (and not different for $i = \pm 4$). Assuming that the grasshopper starts from 0, use the central limit theorem (theorem A.2) to bound the probability that the grasshopper reaches an integer larger than $\left\lceil \sqrt{2T} \right\rceil$ after T steps.
b. Returning to the setting where the grasshopper moves over the integers in the range $\pm B$, try constructing a chain that reaches every state in time linear in B (Hint: Your chain will be nonreversible, and it will require the addition of another variable to the state description.)

Exercise 12.11

Show an example of a Markov chain where the limiting distribution reached via repeated applications of equation (12.20) depends on the initial distribution $P^{(0)}$.

Exercise 12.12⋆

Consider the following two conditions on a Markov chain $\mathcal{T}$:

a. It is possible to get from any state to any state using a positive probability path in the state graph.
b. For each state x, there is a positive probability of transitioning directly from x to x (a self-loop).

a. Show that, for a finite-state Markov chain, these two conditions together imply that $\mathcal{T}$ is regular.
b. Show that regularity of the Markov chain implies condition 1.
c. Show an example of a regular Markov chain that does not satisfy the condition 2.
d. Now let us weaken condition 2, requiring only that there exists a state x with a positive probability of transitioning directly from x to x. Show that this weaker condition and condition 1 together still suffice to ensure regularity.

Exercise 12.13

Show directly from equation (12.21) (without using the detailed balance equation) that the posterior distribution $P(\mathcal{X} \mid \boldsymbol{e})$ is a stationary distribution of the Gibbs chain (equation (12.22)).

Exercise 12.14

Show that any distribution π that satisfies the detailed balance equation, equation (12.24), must be a stationary distribution of $\mathcal{T}$.

Exercise 12.15

Prove theorem 12.5.

Exercise 12.16

Let $Val(\boldsymbol{X})$ be a set of states, and let $\mathcal{T}_1, \ldots, \mathcal{T}_k$ be a set of kernels, each of which satisfies the detailed balance equation relative to some stationary distribution π. Let $p_1, \ldots, p_k$ be any distribution over the indexes $1, \ldots, k$. Prove that the mixture Markov chain $\mathcal{T}$, which at each step takes a step sampled from $\mathcal{T}_i$ with probability p_i, also satisfies the detailed balance equation relative to π.

Exercise 12.17⋆

Let $\mathcal{T}_1, \ldots, \mathcal{T}_k$ be as in exercise 12.16. Consider the aggregate Markov chain $\mathcal{T}$, where each step consists of a sequence of k steps, with step i being sampled from $\mathcal{T}_i$.

a. Provide an example demonstrating that $\mathcal{T}$ may not satisfy the detailed balance equation relative to π.
b. Show that, nevertheless, $\mathcal{T}$ has the stationary distribution π.
c. Provide a multistep kernel using $\mathcal{T}_1, \ldots, \mathcal{T}_k$ that satisfies the detailed balance equation relative to π.

Exercise 12.18

a. Let X be a node in a Markov network $\mathcal{H}$, and let $\boldsymbol{y}$ be an assignment of values to X's Markov blanket $\boldsymbol{Y} = \mathrm{MB}_{\mathcal{H}}(X)$. Provide an efficient algorithm for computing

$$\frac{P(x' \mid \boldsymbol{y})}{P(x \mid \boldsymbol{y})}$$

for $x, x' \in Val(X)$. Your algorithm should involve only local parameters — potentials for cliques involving X.
b. Show how this algorithm applies to Bayesian networks.

Exercise 12.19

Show that Gibbs sampling is a special case of the Metropolis-Hastings algorithm. More precisely, provide a particular proposal distribution Q_i for each local transition $\mathcal{T}^{Q_i}$ that induces precisely the same distribution over the transitions taken as the associated Gibbs transition distribution $\mathcal{T}_i$.

Exercise 12.20⋆

Prove theorem 12.4.

Exercise 12.21

Consider the network of example 12.12, but now assume that we make Z a slightly noisy exclusive or of its parents. That is, with some small probability q, Z is chosen uniformly at random regardless of the values of X and Y, and with probability $1 - q$, Z is the exclusive or of X and Y. Analyze the transitions between states in the Gibbs chain for this network, with the evidence z^1, and particularly the expected time required to transition between different regions.

Exercise 12.22

Consider the same situation as in importance sampling, where we have an unnormalized measure $\tilde{P}(\boldsymbol{X})$ from which it is hard to sample, and a proposal distribution Q which is (hopefully) close to the normalized distribution $P \propto \tilde{P}$, from which we can draw independent samples. Consider a Markov chain where we define

$$\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}') = Q(\boldsymbol{x}') \min\left[1, \frac{w(\boldsymbol{x}')}{w(\boldsymbol{x})}\right]$$

for $\boldsymbol{x}' \neq \boldsymbol{x}$, where $w(\boldsymbol{x})$ is as defined in equation (12.10); we define $\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}) = 1 - \sum_{\boldsymbol{x}' \neq \boldsymbol{x}} \mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}')$. Intuitively, the transition from $\boldsymbol{x}$ to $\boldsymbol{x}'$ selects an independent sample $\boldsymbol{x}'$ from Q and then moves toward it, depending on whether its importance weight is better than that of our current point $\boldsymbol{x}$.

a. Show that $\mathcal{T}$ defines a legal transition model.
b. Show that P is the stationary distribution of $\mathcal{T}$.

Exercise 12.23⋆⋆

Swendson-Wang algorithm

labeling MRF

The *Swendson-Wang algorithm* is a Metropolis-Hastings algorithm for *labeling MRFs*, such as those used in image segmentation (box 4.B), where all variables take values in the same set of labels $l = 1, \ldots, L$. Unlike the standard application of Metropolis-Hastings, this variant takes large steps in the space, which simultaneously flip the values of multiple variables together.

The algorithm proceeds as follows. Let $\mathcal{X}$ be the variables in the network, and $\mathcal{E}$ the set of edges in the pairwise MRF. Let ξ be our current state in the sampling process. We begin by partitioning the variables $\mathcal{X}$ into L disjoint subsets, based on their value in ξ: $\boldsymbol{X}_l = \{X_i \ : \ x_i = l\}$. We now randomly select a subset of the edges in the graph to produce a new set $\mathcal{E}'$: each edge $(i, j) \in \mathcal{E}$ is selected independently, with probability $q_{i,j}$. We now use $\mathcal{E}'$ to partition the graph into connected components. Each set $\boldsymbol{X}_l$ is partitioned into K_l connected components, where a connected component $\boldsymbol{X}_{lk}$ is a maximal subset of nodes $\boldsymbol{X}_{lk} \subseteq \boldsymbol{X}_l$ such that each $X_i, X_j \in \boldsymbol{X}_{lk}$ is connected by a path within $\mathcal{E}'$. The result is a set of connected components:

$$\mathcal{C} = \{\boldsymbol{X}_{lk} \ : \ l = 1, \ldots, L; k = 1, \ldots, K_l\}.$$

We now select a connected component $\boldsymbol{X}_{lk} \in \mathcal{C}$ uniformly at random. Let $q(\boldsymbol{Y} \mid \xi)$ be the probability with which a set $\boldsymbol{Y}$ is selected using this procedure.

We now propose to assign $\boldsymbol{X}_{lk}$ a new label l' with probability $q(l' \mid \xi, \boldsymbol{X}_{lk})$; note that this probability can depend in arbitrary ways on the current state. This proposed move, if accepted, defines a new state ξ', where $X_i' = l'$ for any $X_i \in \boldsymbol{X}_{lk}$, and $X_i' = X_i$ otherwise. Note that this proposed move flips a large number of variables at the same time, and thus it takes much larger steps in the space than a local Gibbs or Metropolis-Hastings sampler for this MRF.

In this exercise, we show how to use this proposal distribution within the Metropolis-Hastings algorithm.

a. Let ξ and ξ' be a pair of states such that:
   - $\boldsymbol{Y}$ forms a connected component in $\mathcal{E}$;
   - the variables in $\boldsymbol{Y}$ all take the value l in ξ;
   - the variables in $\boldsymbol{Y}$ all take the value l' in ξ';
   - all other variables in $\mathcal{X} - \boldsymbol{Y}$ take the same value in ξ and ξ'.

   Show that

$$\frac{q(\boldsymbol{Y} \mid \xi)}{q(\boldsymbol{Y} \mid \xi')} = \frac{\prod_{(i,j) \in \mathcal{E}(\boldsymbol{Y}, \boldsymbol{X}_l - \boldsymbol{Y})} (1 - q_{i,j})}{\prod_{(i,j) \in \mathcal{E}(\boldsymbol{Y}, \boldsymbol{X}'_{l'} - \boldsymbol{Y})} (1 - q_{i,j})}$$

   where: $\boldsymbol{X}_l$ is the set of vertices with label l in ξ, $\boldsymbol{X}'_{l'}$ the set of vertices with label l' in ξ'; and where $\mathcal{E}(\boldsymbol{Y}, \boldsymbol{Z})$ (between two disjoint sets $\boldsymbol{Y}, \boldsymbol{Z}$) is the set of edges connecting nodes in $\boldsymbol{Y}$ to nodes in $\boldsymbol{Z}$.

b. Use this fact to obtain an acceptance probability for the proposed move that respects the detailed-balance equation.

Exercise 12.24⋆⋆

Let us return to the setting of exercise 9.20.

a. Suppose now that you want to sample from $P(\mathcal{X} \mid \text{Possible}(\mathcal{X}) = K)$ using a Gibbs sampling strategy. Why is this a bad idea?

b. Can you devise a Metropolis-Hastings MCMC strategy that generates samples from the correct posterior? Describe the proposal distribution and the acceptance probabilities, and prove that your scheme does sample from the correct posterior. Explain intuitively why your chain is likely to work better than the Gibbs chain. You may assume that $1 < K < N$.

Exercise 12.25⋆⋆

annealed importance sampling

In this exercise we develop the *annealed importance sampling* procedure. Assume that we want to generate samples from a distribution $p(\boldsymbol{x}) \propto f(\boldsymbol{x})$ from which sampling is hard. Assume also that we have a distribution $q(\boldsymbol{x}) \propto g(\boldsymbol{x})$ from which sampling is easy. In principle, we can use q as a proposal distribution for p, and apply importance sampling. However, if q and p are very different, the results are likely to be quite poor. We now construct a sequence of Markov chains that will allow us to incrementally produce a lower-variance importance-sampling estimator.

The technique is as follows. We define a sequence of distributions, $p_0, \ldots, p_k$, where $p_i(\boldsymbol{x}) \propto f_i(\boldsymbol{x})$, and f_i is defined as:

$$f_i(\boldsymbol{x}) = p(\boldsymbol{x})^{\beta_i} q(\boldsymbol{x})^{1-\beta_i},$$

where $1 = \beta_0 > \beta_1 > \ldots > \beta_k = 0$. Note that $p_0 = p$ and $p_k = q$. We assume that we can generate samples from p_k, and that, for each p_i, $i = 1, \ldots, k-1$, we have a Markov chain $\mathcal{T}_i$ whose stationary distribution is p_i. To generate a weighted sample $\boldsymbol{x}, w$ relative to our target distribution p, we follow the following algorithm:

$$\begin{array}{lcll} \boldsymbol{x}_k & \sim & p_k(\boldsymbol{X}) & \\ \boldsymbol{x}_i & \sim & \mathcal{T}_i(\boldsymbol{x}_{i+1} \to \boldsymbol{X}) & i = (k-1), \ldots, 1. \end{array} \tag{12.37}$$

Finally, we define our sample to be $\boldsymbol{x} = \boldsymbol{x}_1$, with weight

$$w = \prod_{i=1}^{k} \frac{f_{i-1}(\boldsymbol{x}_i)}{f_i(\boldsymbol{x}_i)}. \tag{12.38}$$

To prove that these importance weights are correct, we define both a target distribution and a proposal distribution over the larger state space $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. We then show that the importance weights defined in equation (12.38) are correct relative to these distributions over the larger space.

a. Let

$$\mathcal{T}_i^{-1}(\boldsymbol{x} \to \boldsymbol{x}') = \mathcal{T}_i(\boldsymbol{x}' \to \boldsymbol{x}) \frac{f_i(\boldsymbol{x}')}{f_i(\boldsymbol{x})}$$

define the reversal of the transition model defined by $\mathcal{T}_i$. Show that $\mathcal{T}_i^{-1}(\boldsymbol{X} \to \boldsymbol{X}')$ is a valid transition model.

b. Define

$$f^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) = f_0(\boldsymbol{x}_1) \prod_{i=1}^{k-1} \mathcal{T}_i^{-1}(\boldsymbol{x}_i \to \boldsymbol{x}_{i+1}),$$

and define $p^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \propto f^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. Use your answer from above to conclude that $p^*(\boldsymbol{x}_1) = p(\boldsymbol{x}_1)$.

c. Let g^* be the function encoding the joint distribution from which $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ are sampled in the annealed importance sampling procedure equation (12.37). Show that the weight in equation (12.38) can be obtained as

$$\frac{f^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)}{g^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)}.$$

One can show, under certain assumptions, that the variance of the weights obtained by this procedure grows linearly in the dimension n of the number of variables $\boldsymbol{X}$, whereas the variance in a traditional importance sampling procedure grows exponentially in n.

Exercise 12.26

This exercise explores one heuristic approach for deterministic search in a Bayesian network. It is an intermediate method between full-particle search and collapsed-particle search: It uses partial instantiations as particles but does not perform inference on the resulting conditional distribution.

Assume that our goal is to provide upper and lower bounds on the probability of some event $\boldsymbol{y}$ in a Bayesian network $\mathcal{B}$ over $\mathcal{X}$. Let $X_1, \ldots, X_n$ be some topological ordering of $\mathcal{X}$. We enumerate particles that are partial assignments to $\mathcal{X}$, where each partial assignment instantiates some subset $X_1, \ldots, X_k$; note that the set $X_1, \ldots, X_k$ is not an arbitrary subset of $X_1, \ldots, X_n$, but rather the first k variables in the ordering. Different partial assignments may instantiate different prefixes of the variables. We organize these partial assignments in a tree, where each node is labeled with some partial assignment $(x_1, \ldots, x_k)$. The children of a node labeled $(x_1, \ldots, x_k)$ are $(x_1, \ldots, x_k, x_{k+1})$, for each $x_{k+1} \in Val(X_{k+1})$. We can iteratively grow the tree by choosing some leaf in the tree, corresponding to an assignment $(x_1, \ldots, x_k)$, and expanding the tree to include its children $(x_1, \ldots, x_j, x_{k+1})$ for all possible values x_{k+1}.

Consider a particular tree, with a set of leaves $L = \{\ell[1], \ldots, \ell[M]\}$, where each leaf $\ell[m] \in L$ is associated with the assignment $\boldsymbol{x}[m]$ to some subset of variables $\boldsymbol{X}[m]$.

a. Each leaf $\ell[m]$ in the tree defines a particle. Specify the assignment and probability associated with this particle, and describe how we would compute its probability efficiently.

b. Show how to use your probability estimates from part 1 (a) to provide both a lower and an upper bound for $P(\boldsymbol{y})$.

c. Based on your answer from part 1, provide a simple heuristic for choosing the next leaf to expand in the partial search tree.

Exercise 12.27⋆⋆

Consider the application of collapsed Gibbs sampling, where we use a clique tree to manipulate the conditional distribution $\tilde{P}(\boldsymbol{X}_d \mid \boldsymbol{X}_p)$. Develop an algorithm in which, after an initial calibration step, *all* of the variables $X_i \in \boldsymbol{X}_p$ in can be resampled using a single pass over the clique tree. (Hint: Use the algorithm developed in exercise 10.12.)

Exercise 12.28

Consider the setting of example 12.18, where we assume that all grades are observed but none of the I_j or D_k variables are observed. Show how you would use the set of collapsed samples generated in this example to compute the expected value of the number of smart students (i^1) who got a grade of a C (g^3) in an easy class (d^0).

Exercise 12.29⋆

Consider the data-association problem described in box 12.D: We have two sets of objects $\mathcal{U} = \{u_1, \ldots, u_k\}$ and another $\mathcal{V} = \{v_1, \ldots, v_m\}$, and we wish to map $\mathcal{U}$'s to $\mathcal{V}$'s. We have a set of observed features $\boldsymbol{B}_i$ for each object u_i, and a set of hidden attributes $\boldsymbol{A}_j$ for each v_j. We have a prior $P(\boldsymbol{A}_j)$, and a set of factors $\phi_i(\boldsymbol{A}_j, \boldsymbol{B}_i, C_i)$ such that $\phi_i(\boldsymbol{a}_j, \boldsymbol{b}_i, C_i) = 1$ for all $\boldsymbol{a}_j, \boldsymbol{b}_i$ if $C_i \neq j$. The model contains no other potentials.

We wish to compute the posterior over $\boldsymbol{A}_j$ using collapsed Gibbs sampling, where we sample the C_i's but maintain a closed-form posterior over the $\boldsymbol{A}_j$'s. Provide a sampling scheme for this task, showing clearly both the sampling distribution for the C_i variables and the computation of the closed form over the $\boldsymbol{A}_i$ variables given the assignment to the C_i's.

13 *MAP Inference*

13.1 Overview

So far, we have dealt solely with conditional probability queries. However, MAP queries, which we defined in section 2.1.5, are also very useful in a variety of applications. As a reminder, a MAP query aims to find the most likely assignment to all of the (non-evidence) variables. A marginal MAP query aims to find the most likely assignment to a subset of the variables, marginalizing out over the rest.

MAP queries are often used as a way of "filling in" unknown information. For example, we might be trying to diagnose a complex device, and we want to find a single consistent hypothesis about failures in different components that explains the observed behavior. Another example arises when we are trying to decode messages transmitted over a noisy channel. In such cases, the receiver observes a sequence of bits received over the channel, and then it attempts to find the most likely assignment of input bits that could have generated this observation (taking into account the code used and a model of the channel noise). This type of query is much better viewed as a MAP query than as a standard probability query, because we are not interested in the most likely values for the individual bits sent, but rather in the message whose overall probability is highest. A similar phenomenon arises in speech recognition, where we are trying to decode the most likely utterance given the (noisy) acoustic signal; here also we are not interested in the most likely value of individual phonemes uttered.

13.1.1 Computational Complexity

As for the case of conditional probability queries, it is instructive to analyze the computational complexity of the problem. There are many possible ways of formulating the MAP problem as a decision problem. One that is convenient for our purposes is the problem *BN-MAP-DP*, defined as follows:

> Given a Bayesian network $\mathcal{B}$ over $\mathcal{X}$ and a number τ, decide whether there exists an assignment $\boldsymbol{x}$ to $\mathcal{X}$ such that $P(\boldsymbol{x}) > \tau$.

It turns out that a very similar construction to theorem 9.1 can be used to show that the *BN-MAP-DP* problem is also $\mathcal{NP}$-complete.

Theorem 13.1 *The decision problem BN-MAP-DP is $\mathcal{NP}$-complete*

The proof is left as an exercise (exercise 13.1).

We can also define an analogous decision problem *BN-margMAP-DP* for marginal MAP:

> Given a Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a number τ, and a subset $\boldsymbol{Y} \subset \mathcal{X}$, decide whether there exists an assignment $\boldsymbol{y}$ to $\boldsymbol{Y}$ such that $P(\boldsymbol{y}) > \tau$.

Because marginal MAP is a generalization of MAP, we immediately conclude the following:

Corollary 13.1 *The decision problem BN-margMAP-DP is $\mathcal{NP}$-hard.*

However, for the case of marginal MAP, we cannot conclude that *BN-margMAP-DP* is in $\mathcal{NP}$. Intuitively, as we said, the marginal MAP problem involves elements of both maximization and summation, a combination that is significantly harder than either subtask in isolation. In fact, it is possible to show that *BN-margMAP-DP* is complete for a much harder complexity class:

Theorem 13.2 *The decision problem BN-margMAP-DP is complete for $\mathcal{NP}^{\mathcal{PP}}$.*

Defining the complexity class $\mathcal{NP}^{\mathcal{PP}}$ is outside the scope of this book (see section 9.8), but it is generally considered very hard, since it is known to contain the entire polynomial hierarchy, of which $\mathcal{NP}$ is only the first level.

While the "harder" complexity class of the marginal MAP problem indicates that it is more difficult, the implications of this formulation may be somewhat abstract. A more concrete ramification is the following result, which states that the marginal MAP problem is $\mathcal{NP}$-hard even for polytree networks:

Theorem 13.3 *The following decision problem is $\mathcal{NP}$-hard:*

polytree

> *Given a* polytree *Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a subset $\boldsymbol{Y} \subset \mathcal{X}$, and a number τ, decide whether there exists an assignment $\boldsymbol{y}$ to $\boldsymbol{Y}$ such that $P(\boldsymbol{y}) > \tau$.*

We defer the justification for this result to section 13.2.3.

13.1.2 Overview of Solution Methods

As for conditional probability queries, when addressing MAP queries, it is useful to reformulate the joint distribution somewhat more abstractly, as a product of factors. Consider a distribution $P_\Phi(\mathcal{X})$ defined via a set of factors Φ and an unnormalized density $\tilde{P}_\Phi$. We need to compute:

$$\xi^{map} = \arg\max_\xi P_\Phi(\xi) = \arg\max_\xi \frac{1}{Z}\tilde{P}_\Phi(\xi) = \arg\max_\xi \tilde{P}_\Phi(\xi). \tag{13.1}$$

In particular, if $P_\Phi(\mathcal{X}) = P(\mathcal{X} \mid \boldsymbol{e})$, then we aim to maximize $P(\mathcal{X}, \boldsymbol{e})$.

The MAP task goes hand in hand with finding the value of the unnormalized probability of the most likely assignment: $\max_\xi \tilde{P}_\Phi(\xi)$. We note that, given an assignment ξ, we can easily compute its unnormalized probability simply by multiplying all of the factors in Φ, evaluated at ξ. However, we cannot retrieve the *actual* probability of ξ without computing the partition function, a problem that requires that we also solve the sum-product task.

max-product

Because $\tilde{P}_\Phi$ is a product of factors, tasks that involve maximizing $\tilde{P}_\Phi$ are often called *max-*

product inference tasks. Note that we often convert the max-product problem into log-space and maximize $\log \tilde{P}_\Phi$. This logarithm is a sum of factors that correspond to negative energies (see section 4.4.1.2), and hence this version of the problem is often called the *max-sum* problem. It is also common to negate the factors and minimize the sum of the energies for the different potentials; this version is generally called an *energy minimization* problem. The transformation into log-space has several significant advantages. First, it avoids the numerical issues associated with multiplying many small numbers together. More importantly, it transforms the problem into a linear one; as we will see, this transformation allows certain valuable tools to be brought to bear. For consistency with the rest of the book, we mostly use the max-product variant of the problem in the remainder of this chapter. However, all of our discussion carries over with minimal changes to the analogous max-sum (or min-sum) problem: we simply take the logarithm of all factors, and replace factor product steps with factor additions.

max-sum

energy minimization

Many different algorithms, both exact and approximate, have been proposed for addressing the MAP problem. Most obviously, the goal of the MAP task is find an assignment to a set of variables whose score (unnormalized probability) is maximal. Thus, it is an instance of an *optimization problem* (see appendix A.4.1), a class of problems for which many general-purpose solutions have been developed. These methods include heuristic hill-climbing methods (see appendix A.4.2), as well as more specialized optimization methods. Some of these solutions have also been usefully applied to the MAP problem.

There are also many algorithms that are specifically targeted at the max-product (or min-sum) task, and exploit some of its special structure, most notably the connection to the graph representation. A large subset of algorithms operate by first computing a set of factors that are *max-marginals*. Max-marginals are a general notion that can be defined for any function:

Definition 13.1 max-marginal

The max-marginal *of a function f relative to a set of variables $\boldsymbol{Y}$ is:*

$$\mathit{MaxMarg}_f(\boldsymbol{y}) = \max_{\xi\langle \boldsymbol{Y}\rangle = \boldsymbol{y}} f(\xi), \tag{13.2}$$

for any assignment $\boldsymbol{y} \in \mathit{Val}(\boldsymbol{Y})$. ■

For example, the max-marginal $\mathit{MaxMarg}_{\tilde{P}_\Phi}(\boldsymbol{Y})$ is a factor that determines a value for each assignment $\boldsymbol{y}$ to $\boldsymbol{Y}$; this value is the unnormalized probability of the most likely joint assignment consistent with $\boldsymbol{y}$.

A large class of MAP algorithms proceed by first computing an exact or approximate set of max-marginals for all of the variables in $\mathcal{X}$, and then attempting to extract an exact or approximate MAP assignment from these max-marginals. The first phase generally uses techniques such as variable elimination or message passing in clique trees or cluster graphs, algorithms similar to those we applied in the context of sum-product inference. Now, assume we have a set of (exact or approximate) max-marginals $\{\mathit{MaxMarg}_f(X_i)\}_{X_i \in \mathcal{X}}$. A key question is how we use those max-marginals to construct an overall assignment. **As we show, the computation of (approximate) max-marginals allows us to solve a global optimization problem as a set of local optimization problems for individual variables. This task, known as *decoding*, is to construct a joint assignment that locally optimizes each of the beliefs. If we can construct such an assignment, we will see that we can provide guarantees on its (strong local or even global) optimality.** One such setting is when the max-marginals are *unambiguous*: For

decoding max-marginals

unambiguous

each variable X_i, there is a unique x_i^* that maximizes:

$$x_i^* = \arg \max_{x_i \in Val(X_i)} \mathit{MaxMarg}_f(x_i). \tag{13.3}$$

When the max-marginals are unambiguous, identifying the locally optimizing assignment is easy. When they are ambiguous, the solution is nontrivial even for exact max-marginals, and can require an expensive computational procedure in its own right.

The marginal MAP problem appears deceptively similar to the MAP task. Here, we aim to find the assignment whose (conditional) *marginal* probability is maximal. Here, we partition $\mathcal{X}$ into two disjoint subsets, $\mathcal{X} = \boldsymbol{Y} \cup \boldsymbol{W}$, and aim to compute:

$$\boldsymbol{y}^{\textit{m-map}} = \arg\max_{\boldsymbol{y}} P_\Phi(\boldsymbol{y}) = \arg\max_{\boldsymbol{y}} \sum_{\boldsymbol{W}} \tilde{P}_\Phi(\boldsymbol{y}, \boldsymbol{W}). \tag{13.4}$$

Thus, the marginal MAP problem involves both multiplication and summation, a combination that makes the task much more difficult, both theoretically and in practice. In particular, exact inference methods such as variable elimination can be intractable, even in simple networks. And many of the approximate methods that have been developed for MAP queries do not extend easily to marginal MAP. So far, the only effective approximation technique for the marginal MAP task uses a heuristic search over the assignments $\boldsymbol{y}$, while employing some (exact or approximate) sum-product inference over $\boldsymbol{W}$ in the inner loop.

13.2 Variable Elimination for (Marginal) MAP

We begin our discussion with the most basic inference algorithm: variable elimination. We first present the simpler case of pure MAP queries, which turns out to be quite straightforward. We then discuss the issues that arise in marginal MAP queries.

13.2.1 Max-Product Variable Elimination

To gain some intuition for the MAP problem, let us begin with a very simple example.

Example 13.1

Consider the Bayesian network $A \rightarrow B$. Assume we have no evidence, so that our goal is to compute:

$$\begin{aligned} \max_{a,b} P(a,b) &= \max_{a,b} P(a)P(b \mid a) \\ &= \max_a \max_b P(a)P(b \mid a). \end{aligned}$$

Consider any particular value a of A, and let us consider possible completions of that assignment. Among all possible completions, we want to pick one that maximizes the probability:

$$\max_b P(a)P(b \mid a) = P(a) \max_b P(b \mid a).$$

Thus, a necessary condition for our assignment a, b to have the maximum probability is that B must be chosen so as to maximize $P(b \mid a)$. Note that this condition is not sufficient: we must also choose the value of A appropriately; but for any choice of A, we must choose B as described.

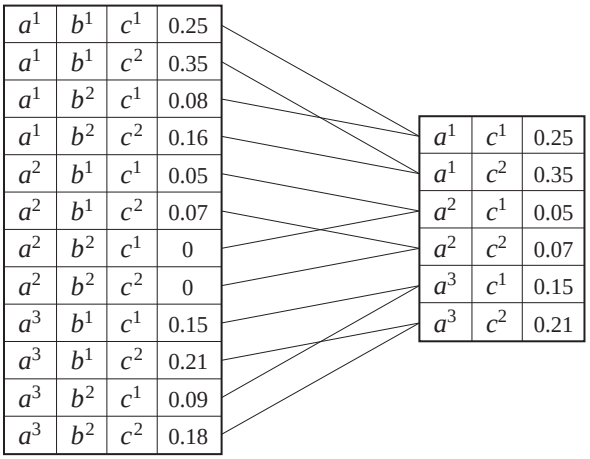


Figure 13.1 Example of the max-marginalization factor operation for variable B

Let $\phi(a)$ denote the internal expression $\max_b P(b \mid a)$. For example, consider the following assignment of parameters:

$$\begin{array}{cc} a^0 & a^1 \\ \hline 0.4 & 0.6 \end{array} \qquad \begin{array}{c|cc} A & b^0 & b^1 \\ \hline a^0 & 0.1 & 0.9 \\ a^1 & 0.55 & 0.45. \end{array} \tag{13.5}$$

In this case, we have that $\phi(a^1) = \max_b P(b \mid a^1) = 0.55$ and $\phi(a^0) = \max_b P(b \mid a^0) = 0.9$.

To compute the max-marginal over A, we now compute:

$$\max_a P(a)\phi(a) = \max[0.4 \cdot 0.9, 0.6 \cdot 0.55] = 0.36.$$

■

As in the case of sum-product queries, we can reinterpret the computation in this example in terms of factors. We define a new operation on factors, as follows:

Definition 13.2 factor maximization

Let $\boldsymbol{X}$ be a set of variables, and $Y \notin \boldsymbol{X}$ a variable. Let $\phi(\boldsymbol{X}, Y)$ be a factor. We define the factor maximization *of Y in ϕ to be factor ψ over $\boldsymbol{X}$ such that:*

$$\psi(\boldsymbol{X}) = \max_Y \phi(\boldsymbol{X}, Y).$$

■

The operation over the factor $P(B \mid A)$ in example 13.1 is performing $\phi(A) = \max_B P(B \mid A)$. Figure 13.1 presents a somewhat larger example.

The key observation is that, like equation (9.6), we can sometimes exchange the order of maximization and product operations: If $X \notin \mathit{Scope}[\phi_1]$, then

$$\max_X(\phi_1 \cdot \phi_2) = \phi_1 \cdot \max_X \phi_2. \tag{13.6}$$

In other words, we can "push in" a maximization operation over factors that do not involve the variable being maximized. A similar property holds for exchanging a maximization with a factor

Step	Variable eliminated	Factors used	Intermediate factor	New factor
1	S	$\phi_S(I,S)$	$\psi_1(I,S)$	$\tau_1(I)$
2	I	$\phi_I(I)$, $\phi_G(G,I,D)$, $\tau_1(I)$	$\psi_2(G,I,D)$	$\tau_2(G,D)$
3	D	$\phi_D(D)$, $\tau_2(G,D)$	$\psi_3(G,D)$	$\tau_3(G)$
4	L	$\phi_L(L,G)$	$\psi_4(L,G)$	$\tau_4(G)$
5	G	$\tau_4(G)$, $\tau_3(G)$	$\psi_5(G)$	$\tau_5(\emptyset)$

Table 13.1 A run of max-product variable elimination

summation operation: If $X \notin \mathit{Scope}[\phi_1]$, then

$$\max_X(\phi_1 + \phi_2) = \phi_1 + \max_X \phi_2. \tag{13.7}$$

max-product variable elimination

This insight leads directly to a *max-product variable elimination* algorithm, which is directly analogous to the algorithm in algorithm 9.1. The difference is that in line 4, we replace the expression $\sum_Z \psi$ with the expression $\max_Z \psi$. The algorithm is shown in algorithm 13.1. The same template also covers max-sum, if we replace product of factors with addition of factors.

If X_i is the final variable in this elimination process, we have maximized all variables other than X_i, so that the resulting factor ϕ_{X_i} is the max-marginal over X_i.

Example 13.2

Consider again our very simple Student *network, shown in figure 3.4. Our goal is to compute the most likely instantiation to the entire network, without evidence. We will use the elimination ordering S, I, D, L, G. Note that, unlike the case of sum-product queries, we have no query variables, so that all variables are eliminated.*

The computation generates the factors shown in table 13.1. For example, the first step would compute $\tau_1(I) = \max_s \phi_S(I,s)$. Specifically, we would get $\tau_1(i^0) = 0.95$ and $\tau_1(i^1) = 0.8$. Note, by contrast, that the same factor computed with summation instead of maximization would give $\tau_1(I) \equiv 1$, as we discussed.

The final factor, $\tau_5(\emptyset)$, is simply a number, whose value is

$$\max_{S,I,D,L,G} P(S,I,D,L,G).$$

For this network, we can verify that the value is 0.184. ■

The factors generated by max-product variable elimination have an identical structure to those generated by the sum-product algorithm using the same ordering. **Thus, our entire analysis of the computational complexity of variable elimination, which we performed for sum-product in section 9.4, applies unchanged. In particular, we can use the same algorithms for finding elimination orderings, and the complexity of the execution is precisely the same induced width as in the sum-product case.** We can also use similar ideas to exploit structure in the CPDs; see, for example, exercise 13.2.

13.2.2 Finding the Most Probable Assignment

decoding

We now tackle the original MAP problem: *decoding*, or finding the most likely assignment itself.

Algorithm 13.1 Variable elimination algorithm for MAP. The algorithm can be used both in its max-product form, as shown, or in its max-sum form, replacing factor product with factor addition.

Procedure Max-Product-VE (
  Φ, // Set of factors over $\boldsymbol{X}$
  $\prec$ // Ordering on $\boldsymbol{X}$
)
Let $X_1, \ldots, X_k$ be an ordering of $\boldsymbol{X}$ such that
  $X_i \prec X_j$ iff $i < j$
  for $i = 1, \ldots, k$
    $(\Phi, \phi_{X_i}) \leftarrow$ Max-Product-Eliminate-Var(Φ, X_i)
  $\boldsymbol{x}^* \leftarrow$ Traceback-MAP($\{\phi_{X_i} : i = 1, \ldots, k\}$)
  return $\boldsymbol{x}^*, \Phi$ // Φ contains the probability of the MAP

Procedure Max-Product-Eliminate-Var (
  Φ, // Set of factors
  Z // Variable to be eliminated
)
  $\Phi' \leftarrow \{\phi \in \Phi : Z \in \mathit{Scope}[\phi]\}$
  $\Phi'' \leftarrow \Phi - \Phi'$
  $\psi \leftarrow \prod_{\phi \in \Phi'} \phi$
  $\tau \leftarrow \max_Z \psi$
  return $(\Phi'' \cup \{\tau\}, \psi)$

Procedure Traceback-MAP (
  $\{\phi_{X_i} : i = 1, \ldots, k\}$
)
  for $i = k, \ldots, 1$
    $\boldsymbol{u}_i \leftarrow (x^*_{i+1}, \ldots, x^*_k)\langle \mathit{Scope}[\phi_{X_i}] - \{X_i\}\rangle$
      // The maximizing assignment to the variables eliminated after X_i
    $x^*_i \leftarrow \arg\max_{x_i} \phi_{X_i}(x_i, \boldsymbol{u}_i)$
      // x^*_i is chosen so as to maximize the corresponding entry in the factor, relative to the previous choices $\boldsymbol{u}_i$
  return $\boldsymbol{x}^*$

As we have discussed, the result of the computation is a max-marginal $\mathit{MaxMarg}_{\tilde{P}_\Phi}(X_i)$ over the final uneliminated variable, X_i. We can now choose the maximizing value x^*_i for X_i. Importantly, from the definition of max-marginals, we are guaranteed that there exists some assignment ξ^* consistent with x^*_i. But how do we construct such an assignment?

We return once again to our simple example:

Example 13.3 *Consider the network of example 13.1, but now assume that we wish to find the actual assignment* $a^*, b^* = \arg\max_{A,B} P(A, B)$. *As we discussed, we first compute the internal maximization*

$\max_b P(a, b)$. *This computation tells us, for each value of* a*, which value of* b *we must choose to complete the assignment in a way that maximizes the probability. In our example, the maximizing value of* B *for* a^1 *is* b^0*, and the maximizing value of* B *for* a^0 *is* b^1*. However, we cannot actually select the value of* B *at this point, since we do not yet know the correct (maximizing) value of* A*. We therefore proceed with the computation of example 13.1, and compute both the max-marginal over* A*,* $\max_a P(a)\phi(a)$*, and the value* a *that maximizes this expression. In this case,* $P(a^1)\phi(a^1) = 0.6 \cdot 0.55 = 0.33$*, and* $P(a^0)\phi(a^0) = 0.4 \cdot 0.9 = 0.36$*. The maximizing value* a^* *of* A *is therefore* a^0*. The key insight is that, given this value of* A*, we can now go back and select the corresponding value of* B *— the one that maximizes* $\phi(a^*)$*. Thus, we obtain that our maximizing assignment is* a^0, b^1*, as expected.* ■

The key intuition in this computation is that, as we eliminate variables, we cannot determine their maximizing value. However, we can determine a "conditional" maximizing value — their maximizing value given the values of the variables that have not yet been eliminated. When we pick the value of the final variable, we can then go back and pick the values of the other variables accordingly. For the last variable eliminated, say X, the factor for the value x contains the probability of the most likely assignment that contains $X = x$. Thus, we correctly select the most likely assignment to X, and therefore to all the other variables. This process is called *traceback* of the solution.

traceback

The algorithm implementing this intuition is shown in algorithm 13.1. Note that the operation in line 2 of Traceback-MAP is well defined, since all of the variables remaining in $Scope[\phi_{X_i}]$ were eliminated after X_i, and hence must be within the set $\{X_{i+1}, \ldots, X_k\}$. We can show that the algorithm returns the MAP:

Theorem 13.4

The algorithm of algorithm 13.1 returns

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x}} \prod_{\phi \in \Phi} \phi,$$

and Φ*, which contains a single factor of empty scope whose value is:*

$$\max_{\boldsymbol{x}} \prod_{\phi \in \Phi} \phi.$$

The proof follows in a straightforward way from the preceding intuitions, and we leave it as an exercise (exercise 13.3).

We note that the traceback procedure is not an expensive one, since it simply involves a linear traversal over the factors defined by variable elimination. In each case, when we select a value x_i^* for a variable X_i in line 2, we are guaranteed that x_i^* is, indeed, a part of a jointly coherent MAP assignment. Thus, we will never need to backtrack and revisit this decision, trying a different value for X_i.

Example 13.4

Returning to example 13.2, we now consider the traceback phase. We begin by computing $g^* = \arg\max_g \psi_5(g)$*. It is important to remember that* g^* *is* not *the value that maximizes* $P(G)$*. It is the value of* G *that participates in the most likely complete assignment to* all *the network variables* $\mathcal{X} = \{S, I, D, L, G\}$*. Given* g^**, we can now compute* $l^* = \arg\max_l \psi_4(g^*, l)$*. The value* l^* *is*

the value of L in the most likely complete assignment to $\mathcal{X}$. We use the same procedure for the remaining variables. Thus,

$$\begin{aligned} d^* &= \arg\max_d \psi_3(g^*, d) \\ i^* &= \arg\max_i \psi_2(g^*, i, d^*) \\ s^* &= \arg\max_s \psi_1(i^*, s). \end{aligned}$$

It is straightforward (albeit somewhat tedious) to verify that the most likely assignment is d^1, i^0, g^3, s^0, l^0, and its probability is (approximately) the value 0.184 that we obtained in the first part of the computation. ■

The additional step of computing the actual assignment does not add significant time complexity to the basic max-product task, since it simply does a second pass over the same set of factors computed in the max-product pass. With an appropriate choice of data structures, this cost can be linear in the number n of variables in the network. The cost in terms of space is a little greater, inasmuch as the MAP pass requires that we store the intermediate results in the max-product computation. However, the total cost is at most a factor of n greater than the cost of the computation without this additional storage.

The algorithm of algorithm 13.1 finds the one assignment of highest probability. This assignment gives us the single most likely explanation of the situation. In many cases, however, we want to consider more than one possible explanation. Thus, a common task is to find the set of the K most likely assignments. This computation can also be performed using the output of a run of variable elimination, but the algorithm is significantly more intricate. (See exercise 13.5 for one simpler case.) An alternative approach is to use one of the search-based algorithms that we discuss in section 13.7.

13.2.3 Variable Elimination for Marginal MAP ⋆

We now turn our attention to the application of variable elimination algorithms to the marginal MAP problem. Recall that our marginal MAP problem can be written as $\arg\max_{\boldsymbol{Y}} \sum_{\boldsymbol{W}} \tilde{P}_\Phi(\boldsymbol{Y}, \boldsymbol{W})$, where $\boldsymbol{Y} \cup \boldsymbol{W} = \mathcal{X}$, so that $\tilde{P}_\Phi(\boldsymbol{Y}, \boldsymbol{W})$ is a product of factors in some set Φ. Thus, our computation has the following *max-sum-product* form:

max-sum-product

$$\max_{\boldsymbol{Y}} \sum_{\boldsymbol{W}} \prod_{\phi \in \Phi} \phi. \tag{13.8}$$

This form immediately suggests a variable elimination algorithm, along the lines of similar algorithms for sum-product and max-product. This algorithm simply puts together the ideas we used for probability queries on one hand and MAP queries on the other. Specifically, the summations and maximizations outside the product can be viewed as operations on factors. Thus, to compute the value of this expression, we simply have to eliminate the variables $\boldsymbol{W}$ by summing them out, and the variables in $\boldsymbol{Y}$ by maximizing them out. When eliminating a variable X, whether by summation or by maximization, we simply multiply all the factors whose scope involves X, and then eliminate X to produce the resulting factor. Our ability to perform this step is justified by the exchangeability of factor summation/maximization and factor product (equation (9.6) and equation (13.6)).

Example 13.5

Consider again the network of figure 3.4, and assume that we wish to find the probability of the most likely instantiation of SAT result and letter quality:

$$\max_{S,L} \sum_{G,I,D} P(I, D, G, S, L).$$

We can perform this computation by eliminating the variables one at a time, as appropriate. Specifically, we perform the following operations:

$$\begin{aligned}
\psi_1(I, G, D) &= \phi_D(D) \cdot \phi_G(G, I, D) \\
\tau_1(I, G) &= \sum_D \psi_1(I, G, D) \\
\\
\psi_2(S, G, I) &= \phi_I(I) \cdot \phi_S(S, I) \cdot \tau_1(I, G) \\
\tau_2(S, G) &= \sum_I \psi_2(S, G, I) \\
\\
\psi_3(S, G, L) &= \tau_2(S, G) \cdot \phi_L(L, G) \\
\tau_3(S, L) &= \sum_G \psi_3(S, G, L) \\
\\
\psi_4(S, L) &= \tau_3(S, L) \\
\tau_4(L) &= \max_S \psi_4(S, L) \\
\\
\psi_5(L) &= \tau_4(L) \\
\tau_5(\emptyset) &= \max_L \psi_5(L).
\end{aligned}$$

Note that the first three factors τ_1, τ_2, τ_3 are generated via the operation of summing out, whereas the last two are generated via the operation of maxing out. ■

This process computes the unnormalized probability of the marginal MAP assignment. We can find the most likely values to the max-variables exactly as we did in the case of MAP: We simply keep track of the factors associated with them, and then we work our way backward to compute the most likely assignment; see exercise 13.4.

Example 13.6

Continuing our example, after completing the different elimination steps, we compute the value $l^* = \arg\max_l \psi_5(L)$. We then compute $s^* = \arg\max_s \psi_4(s, l^*)$. ■

The similarity between this algorithm and the previous variable elimination algorithms we described may naturally lead one to conclude that the computational complexity is also similar. Unfortunately, that is not the case: this process is computationally much more expensive than the corresponding variable elimination process for pure sum-product or pure max-product. The difficulty stems from the fact that we are not free to choose an arbitrary elimination ordering. When summing out variables, we can utilize the fact that the operations of summing out different variables commute. Thus, when performing summing-out operations for sum-product variable

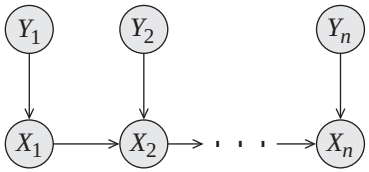


Figure 13.2 A network where a marginal MAP query requires exponential time

elimination, we could sum out the variables in any order. Similarly, we could use the same freedom in the case of max-product elimination. Unfortunately, the max and sum operations do not commute (exercise 13.19). Thus, in order to maintain the correct semantics of marginal MAP queries, as specified in equation (13.4), we *must* perform all the variable summations before we can perform any of the variable maximizations.

As we saw in example 9.1, different elimination orderings can induce very different widths. **When we *constrain* the set of legal elimination orderings, we have a smaller range of possibilities, and even the best elimination ordering consistent with the constraint might have significantly larger width than a good unconstrained ordering.**

constrained elimination ordering

Example 13.7

Consider the network shown in figure 13.2, and assume that we wish to compute

$$\boldsymbol{y}^{m\text{-}map} = \arg \max_{Y_1,\ldots,Y_n} \sum_{X_1,\ldots,X_n} P(Y_1, \ldots, Y_n, X_1, \ldots, X_n).$$

As we discussed, we must first sum out $X_1, \ldots, X_n$, and only then deal with the maximization over the Y_i's. Unfortunately, the factor generated after summing out all of the X_i's contains all of their neighbors, that is, all of the Y_i's. This factor is exponential in n. By contrast, the minimal induced width of this network is 2, so that any probability query (assuming a small number of query variables) or MAP query can be performed on this network in linear time. ■

As we can see, **even on very simple polytree networks, elimination algorithms can require exponential time to solve a marginal MAP query.** One might hope that this blowup is a consequence of the algorithm we use, and that perhaps a more clever algorithm would avoid this problem. Unfortunately, theorem 13.3 shows that this difficulty is unavoidable, and unless $\mathcal{P} = \mathcal{NP}$, some exact marginal MAP computation require exponential time, even in very simple networks. Importantly, however, we must keep in mind that this result does not affect *every* marginal MAP query. Depending on the structure of the network and the choice of maximization variables, the additional cost induced by the constrained elimination ordering may or may not be prohibitive.

Putting aside the issue of computational cost, once we have executed a run of variable elimination for the marginal MAP problem, the task of finding the actual marginal MAP assignment can be addressed using a *traceback* procedure that is directly analogous to Traceback-MAP of algorithm 13.1; we leave the details as an exercise (exercise 13.4).

traceback

Algorithm 13.2 Max-product message computation for MAP

```
Procedure Max-Message (
    i,   // sending clique
    j    // receiving clique
)
1   ψ(C_i) ← ψ_i · ∏_{k∈(Nb_i−{j})} δ_{k→i}
2   τ(S_{i,j}) ← max_{C_i−S_{i,j}} ψ(C_i)
3   return τ(S_{i,j})
```

13.3 Max-Product in Clique Trees

We now extend the ideas used in the MAP variable elimination algorithm to the case of clique trees. As for the case of sum-product, the benefit of the clique tree algorithm is that it uses dynamic programming to compute an entire set of marginals simultaneously. For sum-product, we used clique trees to compute the *sum-marginals* over each of the cliques in our tree. Here, we compute a set of *max-marginals* over each of those cliques.

At this point, one might ask why we want to compute an entire set of max-marginals simultaneously. After all, if our only task is to compute a single MAP assignment, the variable elimination algorithm provides us with a method for doing so. There are two reasons for considering this extension.

First, a set of max-marginals can be a useful indicator for how confident we are in particular components of the MAP assignment. Assume, for example, that our variables are binary-valued, and that the max-marginal for X_1 has $\textit{MaxMarg}(x_1^1) = 3$ and $\textit{MaxMarg}(x_1^0) = 2.95$, whereas the max-marginal for X_2 has $\textit{MaxMarg}(x_2^1) = 3$ and $\textit{MaxMarg}(x_2^0) = 1$. In this case, we know that there is an alternative joint assignment whose probability is very close to the optimum, in which X_1 takes a different value; by contrast, the best alternative assignment in which X_2 takes a different value has a much lower probability. Note that, without knowing the partition function, we cannot determine the actual magnitude of these differences in terms of probability. But we can determine the relative difference between the change in X_1 and the change in X_2.

pseudo-max-marginal

Second, in many cases, an exact solution to the MAP problem via a variable elimination procedure is intractable. In this case, we can use message passing procedures in cluster graphs, similar to the clique tree procedure, to compute *approximate* max-marginals. These *pseudo-max-marginals* can be used for selecting an assignment; while this assignment is not generally the MAP assignment, we can nevertheless provide some guarantees in certain cases. As before, our task has two parts: computing the max-marginals and decoding them to extract a MAP assignment. We describe each of those steps in turn.

13.3.1 Computing Max-Marginals

max-product belief propagation

In the same way that we used dynamic programming to modify the sum-product variable elimination algorithm to the case of clique trees, we can also modify the max-product algorithm to define a *max-product belief propagation* algorithm in clique trees. The resulting algorithm executes precisely the same initialization and overall message scheduling as in the sum-product

max-product message passing

belief propagation algorithm of algorithm 10.2; the only difference is the use of *max-product* rather than sum-product message passing, as shown in algorithm 13.2; as for variable elimination, the procedure has both a max-product and a max-sum variant.

As for sum-product message passing, the algorithm will converge after a single upward and downward pass. After those steps, the resulting clique tree $\mathcal{T}$ will contain the appropriate

max-marginal

max-marginal in every clique.

Proposition 13.1

Consider a run of the max-product clique tree algorithm, where we initialize with a set of factors Φ. Let β_i be a set of beliefs arising from an upward and downward pass of this algorithm. Then for each clique $\boldsymbol{C}_i$ and each assignment $\boldsymbol{c}_i$ to $\boldsymbol{C}_i$, we have that

$$\beta_i(\boldsymbol{c}_i) = \mathit{MaxMarg}_{\tilde{P}_\Phi}(\boldsymbol{c}_i). \tag{13.9}$$

That is, the clique belief contains, for each assignment $\boldsymbol{c}_i$ to the clique variables, the (unnormalized) measure $\tilde{P}_\Phi(\xi)$ of the most likely assignment ξ consistent with $\boldsymbol{c}_i$. The proof is exactly the same as the proof of theorem 10.3 and corollary 10.2 for sum-product clique trees, and so we do not repeat the proof. Note that, because the max-product message passing process does not compute the partition function, we *cannot* derive from these max-marginals the actual probability of any assignment; however, because the partition function is a constant, we can still compare the values associated with different assignments, and therefore compute the assignment ξ that maximizes $\tilde{P}_\Phi(\xi)$.

Because max-product message passing over a clique tree produces max-marginals in every clique, and because max-marginals must agree, it follows that any two adjacent cliques must agree on their sepset:

$$\max_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i = \max_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j = \mu_{i,j}(\boldsymbol{S}_{i,j}). \tag{13.10}$$

max-calibrated

In this case, the clusters are said to be *max-calibrated*. We say that a clique tree is *max-calibrated* if all pairs of adjacent cliques are max-calibrated.

Corollary 13.2

The beliefs in a clique tree resulting from an upward and downward pass of the max-product clique tree algorithm are max-calibrated.

Example 13.8

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 max-marginal beliefs for the clique and sepset for this example are shown in figure 13.3. We can easily confirm that the clique tree is calibrated. ■

max-product belief update

We can also define a *max-product belief update* message passing algorithm that is entirely analogous to the belief update variant of sum-product message passing. In particular, in line 1 of algorithm 10.3, we simply replace the summation with the maximization operation:

$$\sigma_{i \to j} \leftarrow \max_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i.$$

The remainder of the algorithm remains completely unchanged. As in the sum-product case, the max-product belief propagation algorithm and the max-product belief update algorithm