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

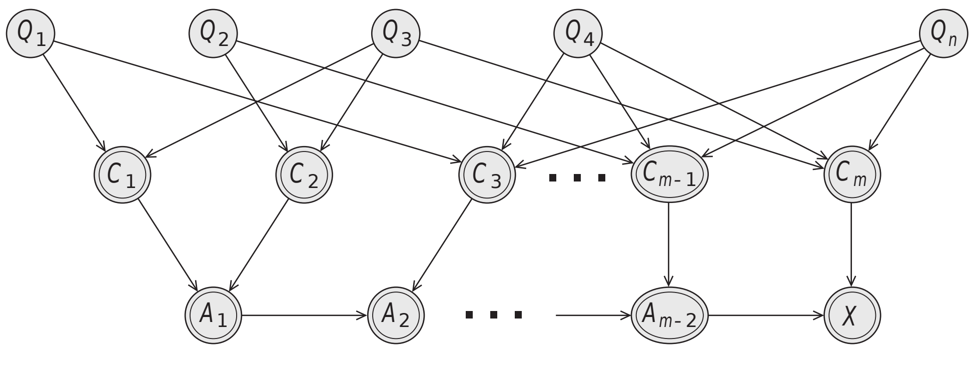


Figure 9.1 An outline of the network structure used in the reduction of 3-SAT to Bayesian network inference.

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:*

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

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 $A \to B \to C \to D$. 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

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

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

Figure 9.2 Computing $P(D)$ by summing over the joint distribution for a chain $A \to B \to C \to D$; 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:

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

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.

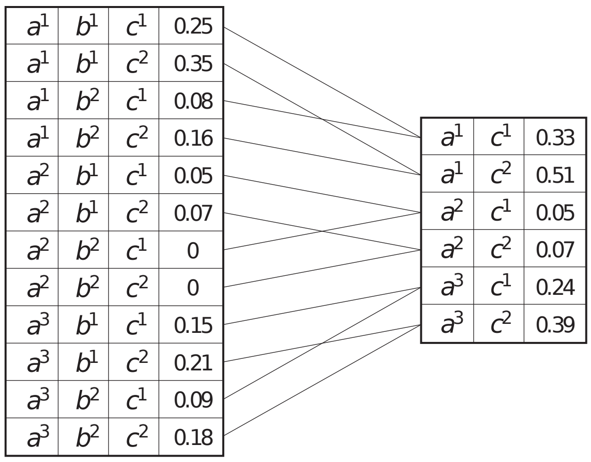
To summarize, the two ideas that help us address the exponential blowup of the joint distribution are:

- **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 $\phi : Val(\boldsymbol{X}) \mapsto I\!R$. 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 (
      $\Phi$,   // Set of factors
      $\boldsymbol{Z}$,   // Set of variables to be eliminated
      $\prec$   // Ordering on $\boldsymbol{Z}$
    )
1     Let $Z_1, \ldots, Z_k$ be an ordering of $\boldsymbol{Z}$ such that
2       $Z_i \prec Z_j$ if and only if $i < j$
3     for $i = 1, \ldots, k$
4       $\Phi \leftarrow$ Sum-Product-Eliminate-Var($\Phi, Z_i$)
5     $\phi^* \leftarrow \prod_{\phi \in \Phi} \phi$
6     return $\phi^*$

    Procedure Sum-Product-Eliminate-Var (
      $\Phi$,   // Set of factors
      $Z$   // Variable to be eliminated
    )
1     $\Phi' \leftarrow \{\phi \in \Phi : Z \in Scope[\phi]\}$
2     $\Phi'' \leftarrow \Phi - \Phi'$
3     $\psi \leftarrow \prod_{\phi \in \Phi'} \phi$
4     $\tau \leftarrow \sum_Z \psi$
5     return $\Phi'' \cup \{\tau\}$
```

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

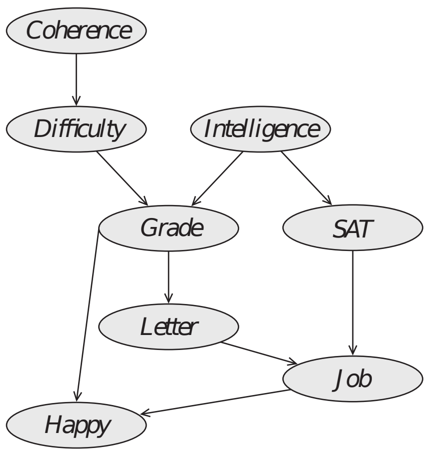


Figure 9.8 The Extended-Student **Bayesian network**

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*

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

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

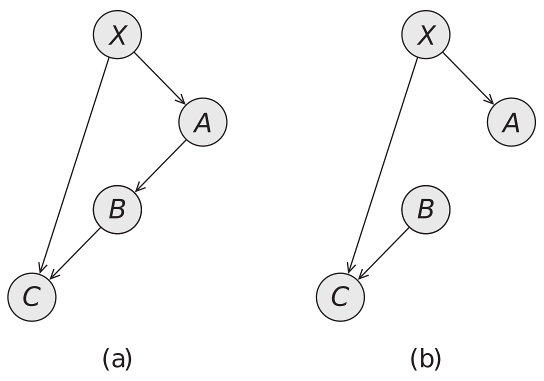


Figure 9.9 Understanding intermediate factors in variable elimination as conditional probabilities: (a) A Bayesian network where elimination does not lead to factors that have an interpretation as conditional probabilities. (b) A different Bayesian network where the resulting factor does correspond to 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 $\alpha, \phi^*$
```

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

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

sensitivity analysis

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).*

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

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

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 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 $X_i, X_j \in$ 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 $\boldsymbol{X}$ = 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 $\boldsymbol{X} = Scope[\Phi_{\boldsymbol{e}}] = \mathcal{X} - \boldsymbol{E}$. 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

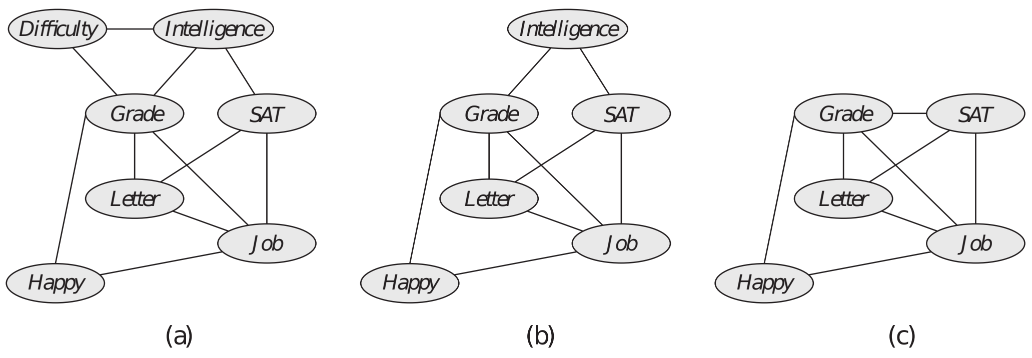


Figure 9.10 Variable elimination as graph transformation in the Student **example,** using the elimination order of table 9.1: (a) after eliminating C; (b) after eliminating D; (c) after eliminating I.

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:

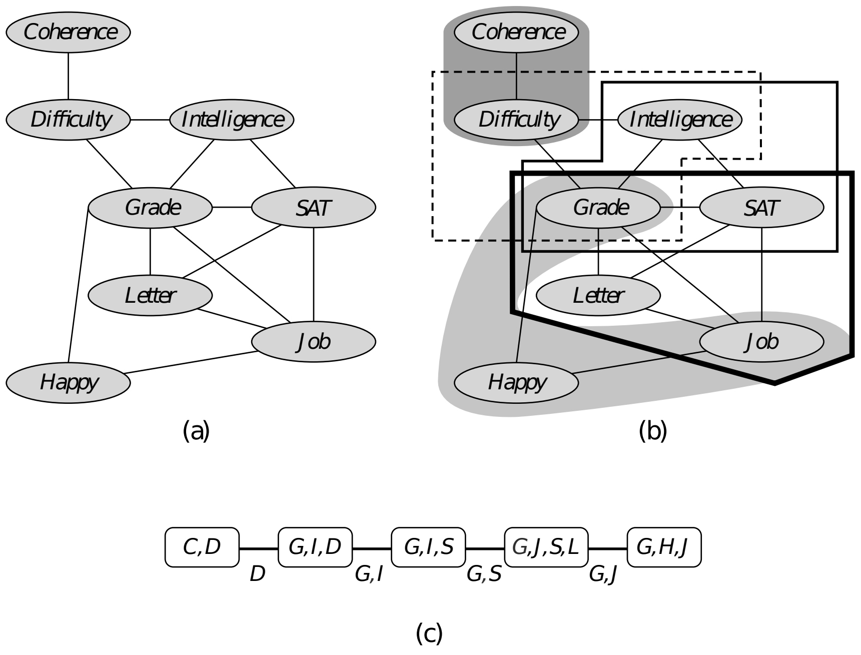


Figure 9.11 Induced graph and clique tree for the Student **example.** (a) Induced graph for variable elimination in the Student example, using the elimination order of table 9.1. (b) Cliques in the induced graph: $\{C, D\}$, $\{D, I, G\}$, $\{G, I, S\}$, $\{G, J, S, L\}$, and $\{G, H, J\}$. (c) Clique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

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

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

induced width

tree-width

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 $\prec$. 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 (
      $\mathcal{H}$   // An undirected graph over $\mathcal{X}$
   )
1     Initialize all nodes in $\mathcal{X}$ as unmarked
2     for $k = |\mathcal{X}| \ldots 1$
3       $X \leftarrow$ unmarked variable in $\mathcal{X}$ with largest number of marked neighbors
4       $\pi(X) \leftarrow k$
5       Mark $X$
6     return $\pi$
```

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 $H \to D$.* ■

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 (
        $\mathcal{H}$    // An undirected graph over $\mathcal{X}$ ,
        $s$    // An evaluation metric
    )
1     Initialize all nodes in $\mathcal{X}$ as unmarked
2     for $k = 1 \ldots |\mathcal{X}|$
3         Select an unmarked variable $X \in \mathcal{X}$ that minimizes $s(\mathcal{H}, X)$
4         $\pi(X) \leftarrow k$
5         Introduce edges in $\mathcal{H}$ between all neighbors of $X$
6         Mark $X$
7     return $\pi$
```

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.

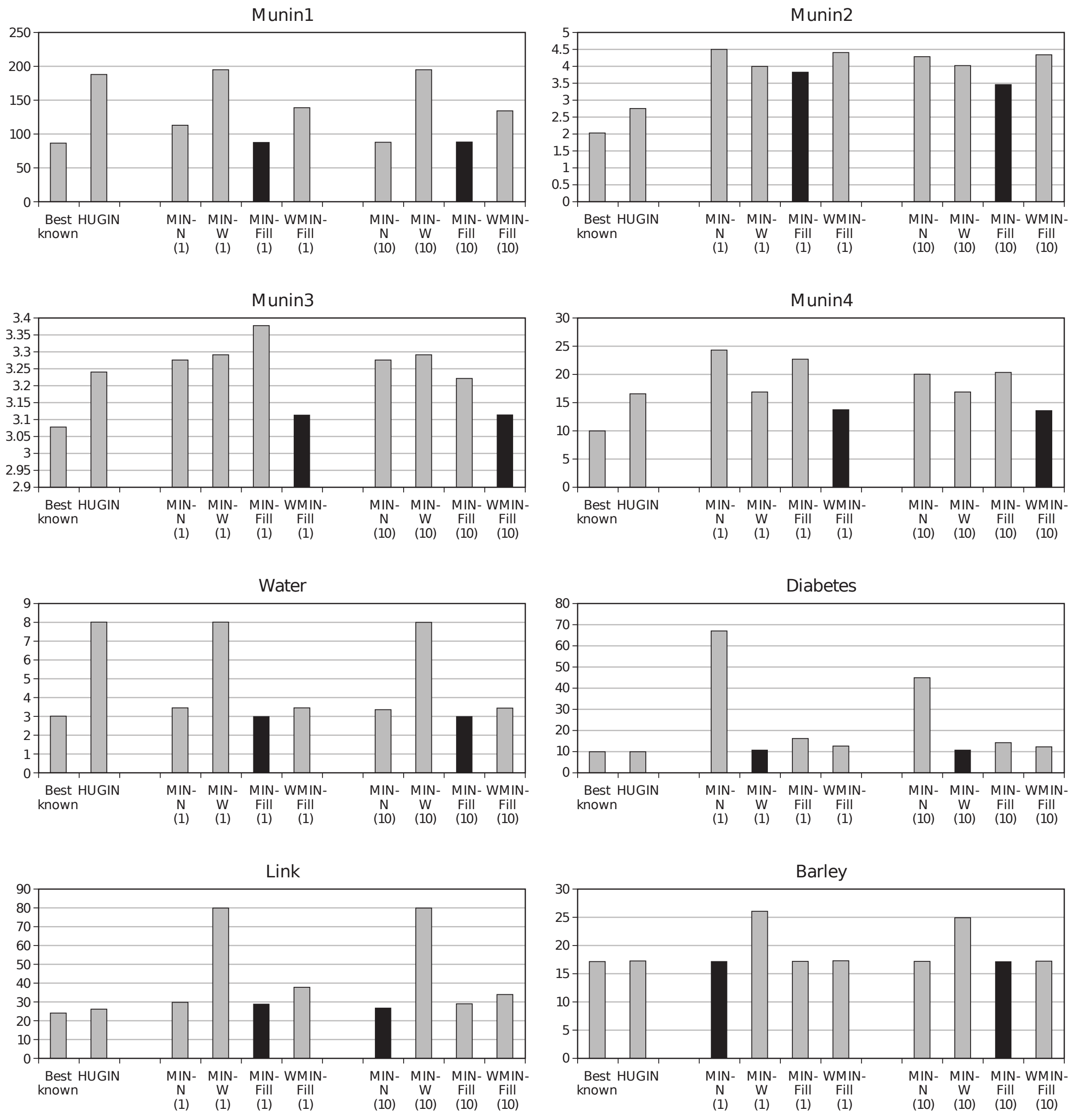


Figure 9.C.1 — Comparison of algorithms for selecting variable elimination ordering. Computational cost of variable elimination inference in a range of benchmark networks, obtained by various algorithms for selecting an elimination ordering. The cost is measured as the size of the factors generated during 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 (
     $\Phi$,   // 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       $\Phi_{\boldsymbol{u}} \leftarrow \{\phi[\boldsymbol{U} = \boldsymbol{u}] \ : \ \phi \in \Phi\}$
3       Construct $\mathcal{H}_{\Phi_{\boldsymbol{u}}}$
4       $(\alpha_{\boldsymbol{u}}, \phi_{\boldsymbol{u}}(\boldsymbol{Y})) \leftarrow$ 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 $\boldsymbol{u} \in \mathit{Val}(\boldsymbol{U})$. The cost of this computation is $O(|\mathit{Val}(\boldsymbol{U})|)$, 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 $\mathcal{H}_{\Phi_{\boldsymbol{u}}}$ 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*

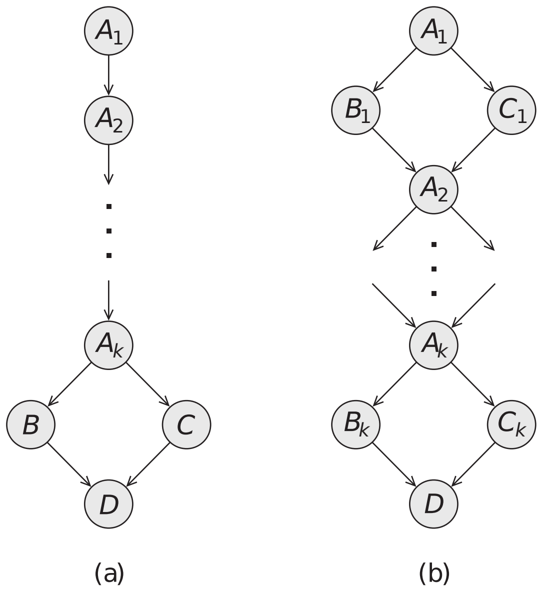


Figure 9.12 Networks where conditioning performs unnecessary computation

to cut the single loop in the network. In this case, we would perform the entire elimination of the chain $A_1 \to \ldots \to A_{k-1}$ 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.

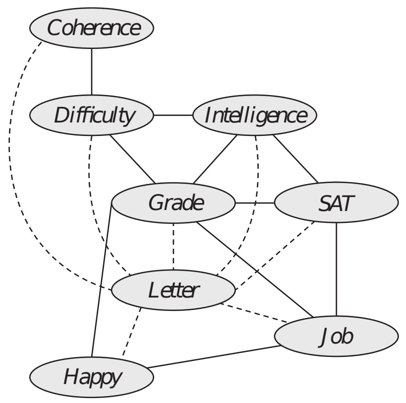


Figure 9.13 Induced graph for the Student **example using both conditioning and elimination:** we condition on L and eliminate the remaining variables using the ordering C, D, I, H, G, S.

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 $A_1 \to \ldots \to A_{k-1}$ 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

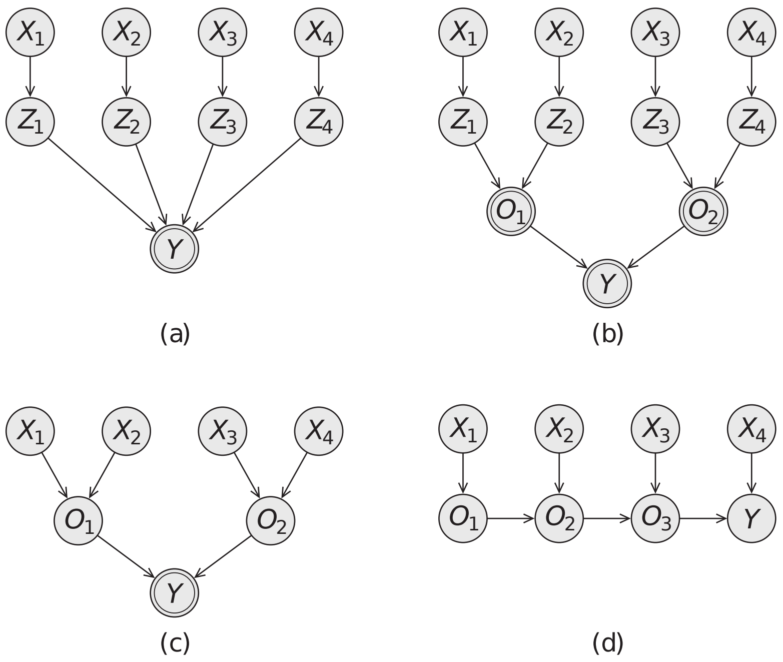


Figure 9.14 Different decompositions for a noisy-or CPD: (a) The standard decomposition of a noisy-or. (b) A tree decomposition of the deterministic-or. (c) A tree-based decomposition of the noisy-or. (d) A chain-based decomposition of the noisy-or.

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

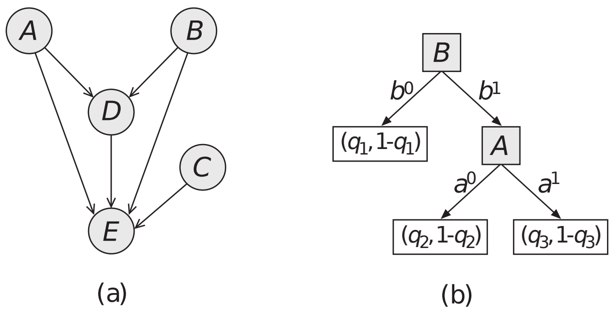


Figure 9.15 A Bayesian network with rule-based structure: (a) the network structure; (b) the CPD for the variable D.

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:*

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

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:

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

■

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 (
      $\rho = \langle \boldsymbol{c}; p \rangle$,   // Rule to be split
      $\boldsymbol{c}'$   // Context to split on
    )
1     if $\boldsymbol{c} \not\sim \boldsymbol{c}'$ then return $\rho$
2     if $Scope[\boldsymbol{c}] \subseteq Scope[\boldsymbol{c}']$ then return $\rho$
3     Select $Y \in Scope[\boldsymbol{c}'] - Scope[\boldsymbol{c}]$
4     $\mathcal{R} \leftarrow$ $Split(\rho \angle Y)$
5     $\mathcal{R}' \leftarrow$ $\cup_{\rho'' \in \mathcal{R}}$ Rule-Split($\rho'', \boldsymbol{c}'$)
6     return $\mathcal{R}'$
```

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

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

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:

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

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          $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \{\rho_1, \rho_2\} \cup$ Rule-Split$(\rho_1, \boldsymbol{c}_2) \cup$ 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          $\mathcal{R}^+ \leftarrow \mathcal{R}^+ - \{\rho_1, \rho_2\} \cup$ Rule-Split$(\rho_1, \boldsymbol{c}_2) \cup$ 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:

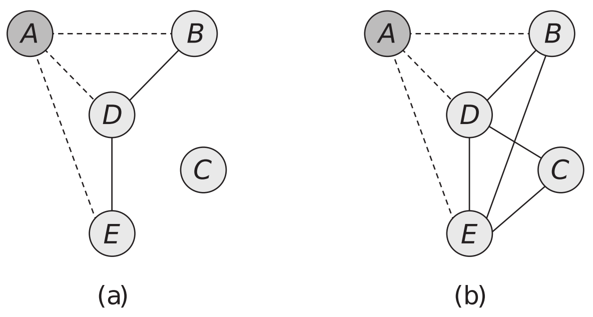


Figure 9.16 Conditioning a Bayesian network whose CPDs have CSI: (a) conditioning on a^0; (b) conditioning on a^1.

The probability of a context $\boldsymbol{c}$ such that *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 $\mathcal{B}$:

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