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 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 partial MAP query). However, this approach to the inference problem is not very satisfactory, since it returns us to the exponential blowup of the joint distribution that the graphical model representation was precisely designed to avoid.

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

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

9.1.1 Analysis of Exact Inference

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

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

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

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

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

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

3-SAT

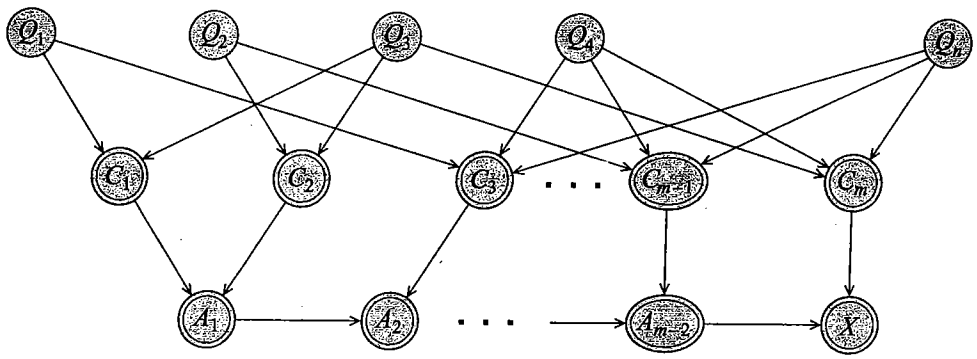

Figure 9.1 An outline of the network structure used in the reduction of 3-SAT to Bayesian network inference.

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

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

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

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

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

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

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

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

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

9.1.2 Analysis of Approximate Inference

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

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

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

Definition 9.1 absolute error

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

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

■

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

Definition 9.2 relative error

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

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

■

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

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

This result is straightforward for the case of relative error.

Theorem 9.3

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

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

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

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

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

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

Theorem 9.4

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

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

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

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

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

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

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

9.2 Variable Elimination: The Basic Ideas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

Figure 9.3 The first transformation on the sum of figure 9.2

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

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

Figure 9.4 The second transformation on the sum of figure 9.2

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

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

Figure 9.5 The third transformation on the sum of figure 9.2

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

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

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 two 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 12. By comparison, generating the joint distribution requires $16 \cdot 3 = 48$

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

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

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

We push in the first summation, resulting in

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

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

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

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

dynamic programming

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

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

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

9.3 Variable Elimination

factor

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

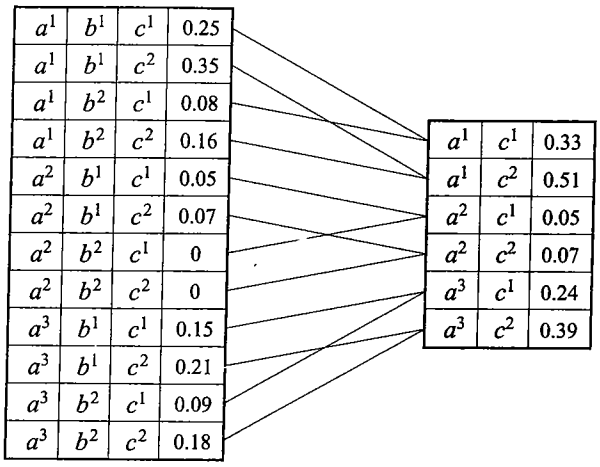

Figure 9.7 Example of factor marginalization: summing out B.

9.3.1 Basic Elimination

9.3.1.1 Factor Marginalization

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

Definition 9.3 factor marginalization

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

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

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

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

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

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

Algorithm 9.1 Sum-product variable elimination algorithm

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

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

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

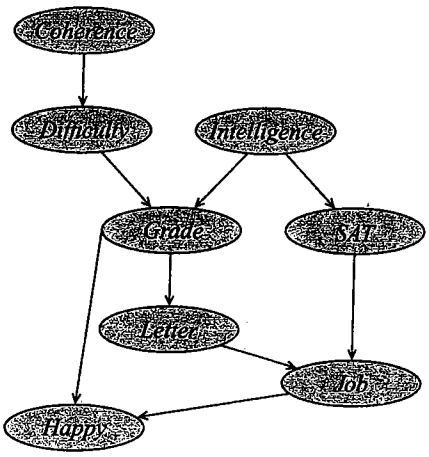

Figure 9.8 The Extended-Student **Bayesian network**

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

Example 9.1

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

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

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

1. *Eliminating C: We compute the factors*

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

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

D. Hence, we now compute:

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

3. *Eliminating I: We compute the factors*

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

4. *Eliminating H: We compute the factors*

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

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

5. *Eliminating G: We compute the factors*

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

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	$\tau_5(D,J)$, $\phi_D(D,C)$	D,J,C	$\tau_6(D,J)$
7	D	$\tau_6(D,J)$	D,J	$\tau_7(J)$

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

9.3.1.3 Semantics of Factors

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

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

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

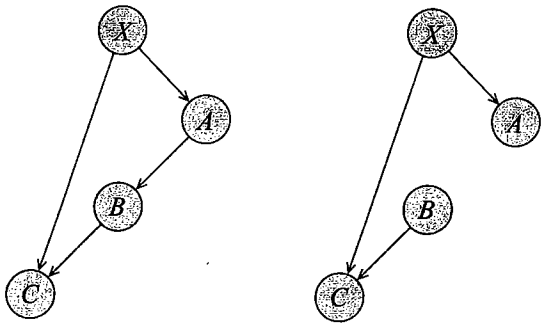

Figure 9.9 Understanding intermediate factors in variable elimination as conditional probabilities: (a) A Bayesian network where elimination does not lead to factors that have an interpretation as conditional probabilities. (b) A different Bayesian network where the resulting factor does correspond to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

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

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

9.3.2 Dealing with Evidence

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

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

factor reduction

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

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

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

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

Step	Variable eliminated	Factors used	Variables involved	New factor
1'	C	$\phi_C(C)$, $\phi_D(D, C)$	C, D	$\tau'_1(D)$
2'	D	$\phi_G[I = i^1](G, D)$, $\phi_I[I \doteq i^1]()$, $\tau'_1(D)$	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 $Y_i \neq y_i$;*
- *$Y_i \notin \boldsymbol{Y}$, define $\lambda^{\boldsymbol{y}}_{y_i} = 1$ for all $y_i \in Val(Y_i)$.*

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

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

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

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

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

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

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

sensitivity analysis

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

9.4 Complexity and Graph Structure: Variable Elimination

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

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

9.4.1 Simple Analysis

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

The algorithm consists of a set of elimination steps, where, in each step, the algorithm picks a variable X_i, then multiplies all factors involving that variable. The result is a single large factor ψ. The variable then gets summed out of ψ_i, resulting in a new factor τ_i whose scope is the scope of ψ minus X. 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 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. 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 one 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 $\Phi[e]$ 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

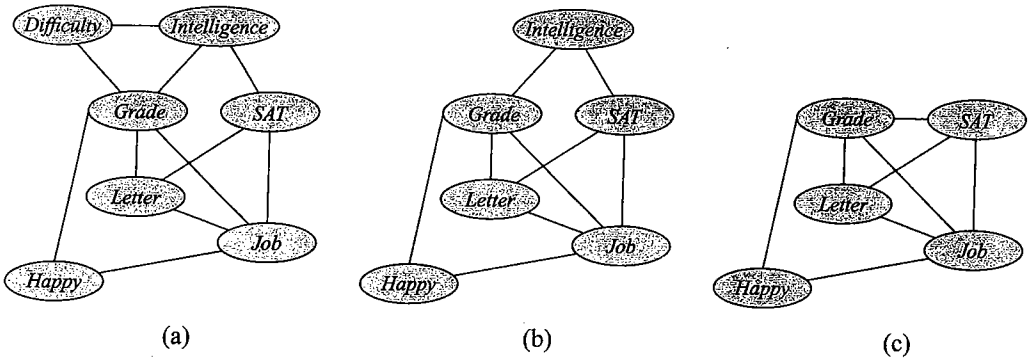

Figure 9.10 Variable elimination as graph transformation in the Student **example,** using the elimination order of table 9.1: (b) after eliminating C; (c) after eliminating D; (d) after eliminating I.

$\boldsymbol{Y}$ 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

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. For clarity, the figure still contains variables and edges that are removed, but marked with dashed lines. 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.* ■

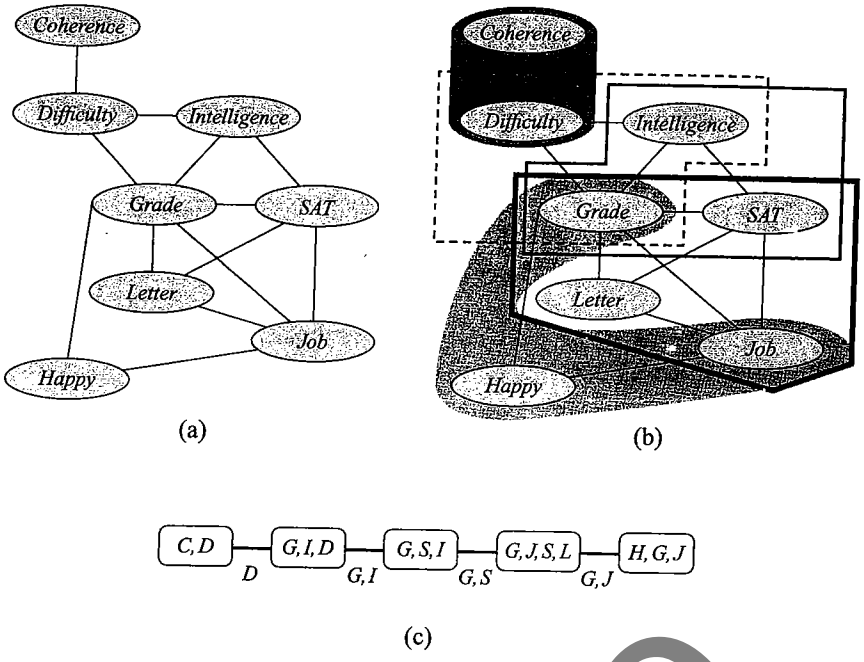

Figure 9.11 Induced graph and clique tree for the Student **example.** (a) Induced graph for variable elimination in the Student example, using the elimination order of table 9.1. (b) Cliques in the induced graph: $\{C, D\}$, $\{D, I, G\}$, $\{I, G, S\}$, $\{G, J, S, L\}$, and $\{G, J, H\}$. (c) Clique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

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 $\mathcal{G}_{\prec}$ and factors ψ is, in fact, much tighter:

Theorem 9.6

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

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

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

To prove the second statement, consider some maximal clique $\boldsymbol{Y} = \{Y_1, \ldots, Y_k\}$. Assume, without loss of generality, that Y_1 is the first of the variables in $\boldsymbol{Y}$ in the ordering $\prec$, and is

therefore the first among this set to be eliminated. Since $\boldsymbol{Y}$ is a clique, there is an edge from Y_1 to each other Y_i. Note that, once Y_1 is eliminated, it can appear in no more factors, so there can be no new edges added to it. Hence, the edges involving Y_1 were added prior to this point in the computation. The existence of an edge between Y_1 and Y_i therefore implies that, at this point, there is a factor containing both Y_1 and Y_i. When Y_1 is eliminated, all these factors must be multiplied. Therefore, the product step results in a factor ψ that contains all of $Y_1, Y_2, \ldots, Y_k$. Note that this factor can contain no other variables; if it did, these variables would also have an edge to all of $Y_1, \ldots, Y_k$, so that $Y_1, \ldots, Y_k$ would not constitute a maximal connected subgraph. ■

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

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

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

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

Definition 9.6 *We define the* width *of an induced graph to be the number of nodes in the largest clique in the graph minus 1. We define the* induced width *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 $w^*_{\mathcal{K}} = \min_{\prec} w(\mathcal{I}_{\mathcal{K},\prec})$.* ■

induced width

tree-width

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 $\boldsymbol{C}_i$ but not in its neighbor. Let $\mathcal{H}'$ be the graph obtained by eliminating X_i. Because X_i belongs only to the clique $\boldsymbol{C}_k$, its neighbors are precisely $\boldsymbol{C}_k - \{X_i\}$. Because all of them are also in $\boldsymbol{C}_k$, they are connected to each other. Hence, eliminating X_i introduces no fill edges. Because $\mathcal{H}'$ is also chordal, we can now apply the inductive hypothesis, proving the result. ■

Algorithm 9.3 Maximum cardinality search for constructing an elimination ordering

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

Example 9.2

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

maximum cardinality

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

Example 9.3

Consider applying Max-Cardinality *to the chordal graph of figure 9.11. Assume that the first node selected is S. The second node selected must be one of S's neighbors, say J. The nodes that have 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, D, C, H. Importantly, the actual elimination ordering proceeds in reverse. Thus, we first eliminate H, then C, D, 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 H from a leaf clique, then C, D, then G from the clique $\{G, I, D\}$, which is now (following the elimination of C, D) a leaf.* ■

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

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

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

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

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

triangulation

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

polytree

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

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

9.4.3.2 Minimum Fill/Size/Weight Search

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

Algorithm 9.4 Greedy search for constructing an elimination ordering

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

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

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

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

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

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

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

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

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

9.5 Conditioning ⋆

conditioning

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

9.5.1 The Conditioning Algorithm

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