Lecture Notes #3: Inference with Bayesian Networks*

Daphne Koller
Stanford University

Nir Friedman
Hebrew University

November 22, 2000

1 The Basic Problem

A Bayesian network is a representation of a probability distribution. Therefore, at least in principle, it contains within it the information required to answer any question that is applicable to a distribution. Indeed, as we will see throughout the course of the book, there are many questions that are of interest. In this chapter, we focus only on the most common ones.

1.1 Query types

The type of query that is most commonly our goal is the standard *conditional probability* query. Such a query consists of two parts:

- the *evidence*: a subset $\mathbf{Z}$ of random variables in the network, and an instantiation $\mathbf{z}$ to these variables;
- the *query*: a subset $\mathbf{Y}$ of random variable in the network.

Our task is to compute

$$P(\mathbf{Y} \mid \mathbf{Z} = \mathbf{z}),$$

i.e., the probability distribution over the values $\mathbf{y}$ of $\mathbf{Y}$, conditioned on the fact that $\mathbf{Z} = \mathbf{z}$. By the definition of conditional probability, we know that

$$P(\mathbf{Y} \mid \mathbf{Z} = \mathbf{z}) = \frac{P(\mathbf{Y}, \mathbf{z})}{P(\mathbf{z})} .$$

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

$$P(\mathbf{y}, \mathbf{z}) = \sum_{\mathbf{w}} P(\mathbf{y}, \mathbf{z}, \mathbf{w})$$

As $\mathbf{Y}, \mathbf{Z}, \mathbf{W}$ are all of the network variables, the expression $P(\mathbf{y}, \mathbf{z}, \mathbf{w})$ in the summation is an entry in the joint distribution. Its value can therefore be computed using the Chain Rule for Bayesian networks.

*this is based on draft of a text book, some of the references and citations are missing

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

$$P(\mathbf{z}) = \sum_{\mathbf{y}} P(\mathbf{y}, \mathbf{z}).$$

If we then divide each $P(\mathbf{y}, \mathbf{z})$ by $P(\mathbf{z})$, we get the desired conditional probability. Note that this process corresponds to taking the vector of marginal probabilities $P(\mathbf{y}^0, \mathbf{z}), \ldots, P(\mathbf{y}^k, \mathbf{z})$ (where k is the number of values $\mathbf{Y}$ can take) and normalizing the entries to sum to 1.

Another type of query that often arises is that of finding the *most probable explanation (MPE)*. As for a conditional probability queries, we have evidence $\mathbf{Z} = \mathbf{z}$. Here, our goal is to find the instantiation of all of the remaining variables that has the highest probability given the evidence. Letting $\mathbf{W} = \mathbf{X} - \mathbf{Z}$, we define

$$\text{MPE}(\mathbf{W} \mid \mathbf{z}) = \text{argmax}_{\mathbf{w}} P(\mathbf{w}, \mathbf{z})$$

where, in general, $\text{argmax}_x f(x)$ represents the value of x for which $f(x)$ is maximal.

Finally, the *maximum a posteriori (MAP)* type of query combines features of both of the previous types. Given evidence $\mathbf{Z} = \mathbf{z}$ and a subset of the variables $\mathbf{Y}$, the goal is to find the most likely assignment only to the variables in $\mathbf{Y}$ given the evidence $\mathbf{Z} = \mathbf{z}$:

$$\text{MAP}(\mathbf{Y} \mid \mathbf{z}) = \text{argmax}_{\mathbf{y}} P(\mathbf{y} \mid \mathbf{z})$$

MPE and MAP queries are often used as way of “filling in” unknown information. For example, suppose one wants to describe the “prototypical” patient that suffers from a certain disease. The MPE assignment, given the evidence that the patient suffers from the disease, describes the most like value combination in this case. If we want to consider the most likely symptoms the such a patient will complain on, we compute the MAP assignment to the symptom variables, given the observation that patient suffers from the disease.

These queries are often used in decoding or understanding of signals. For example, in decoding msgs over a noisy channels, one often uses a redundant code. The receiver observes a sequence of bits and then attempts to find the most likely assignment of input bits that could have generated this observation (taking into account the coders encoding mechanism and a model of the channel noise). This is essentially a MAP query. Another application of MPE queries is in speech recognition. A speech recognition system has a model of the sound signal that can be generated by different phonemes (basic units of spoken language). Given a voice signal, the system attempts to reconstruct the sequence of phonemes that is most likely to generate this signal. (We will discuss in more detail the type of networks used in this application in later chapters.)

It is important to note that MPE/MPE assignments are not always in agreement with each other. Thus, most likely assignment $\text{MAP}(Y_1 \mid \mathbf{z})$ might be completely different from the assignment to Y_1 in $\text{MAP}(\{Y_1, Y_2\} \mid \mathbf{z})$. Indeed, consider even a two node chain $A \to B$, with no evidence, and assume that the CPD of A has $P(a^1) = 0.6$, $P(a^0) = 0.4$. Further, assume that the CPD of B has $P(b^1 \mid a^1) = 0.5$ whereas $P(b^1 \mid a^0) = 0.9$. Then $\text{MAP}(A) = a^1$, whereas $\text{MAP}(A, B) = (a^0, b^1)$. To understand this example, note that a^1 is more likely than a^0. However, both values of B have the same probably given a^1. Thus, the most likely assignment contain a^1 has probability $0.6 \times 0.5 = 0.3$. On the other hand, the distribution over values of B is more skewed given a^0, and the most likely assignment has the probability $0.4 \times 0.9 = 0.36$.

1.2 Complexity

In principle, a Bayesian network can be used to answer all of the query types described above. We simply generate the joint distribution, and exhaustively sum out the joint (in the case of a

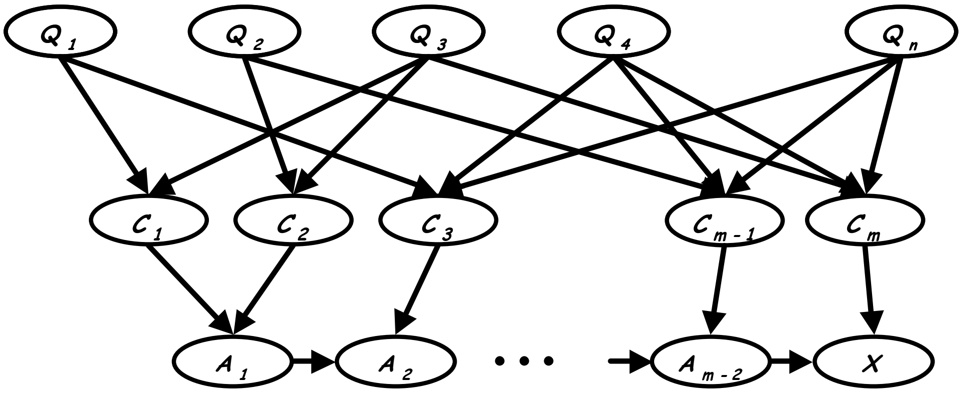

Figure 1: An outline of the network structure used in the reduction of 3-SAT to Bayesian network inference.

conditional probability query), search for the most likely entry (in the case of a MPE query), or both (in the case of a MAP query). However, this approach to the inference problem is not very satisfactory, as it returns us to the exponential blowup of the joint distribution that the Bayesian network representation was precisely designed to avoid.

In the this section, we show that, in a sense, this blowup is unavoidable in the worst case. However, we then devote the remainder of this chapter and the next two chapters to showing that the structure of a Bayesian network can be exploited by our inference algorithms to allow very efficient inference even for some very large networks.

Given the expressive power of Bayesian networks, it should not be surprising that inference, in the worst case, is hard. We now make this statement more formal.

Recall that a language $\mathcal{L}$ is in the class NP if there is a *nondeterministic Turing machine* that can decide in polynomial number of steps whether a string belongs to the language. A language $\mathcal{L}$ is NP-hard if for every language $\mathcal{L}'$ in NP, there is a polynomial transformation of strings such that a string belongs to $\mathcal{L}'$ if and only if the transformed instance belongs to $\mathcal{L}$. Thus, machine that decide if strings belong to $\mathcal{L}$ can be used to decide if strings belong to any language in NP with at most polynomial overhead. We are usually interested in decision problems, these correspond to language that contain all the sequences that correspond to instances for which the decision is positive (e.g., string that encode satisfiable 3-CNF formulas).

The question of complexity touches on how we encode a Bayesian networks. Without going in to too much detail, we can assume that the encoding specifies the DAG structure and the CPDs. For the following theorem we assume the worst case representation of a CPD as a full table of size $|\mathit{Val}(\{X_i\} \cup \mathbf{Pa}_i)|$.

Theorem 1.1: *The following problem is NP-hard:*

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

Note that this result holds even for the marginal distribution of a single variable, without evidence.

Proof: The prove NP-hardness, we need to show that, if we can answer this question, we can use that as a subroutine to answer questions in a class of problems that is known to be NP-hard. We will use a reduction from the standard 3-SAT problem. Figure 1 illustrates the structure of the network constructed in this reduction.

Consider a 3-SAT instance ϕ over the propositional variables $q_1, \ldots, q_n$; ϕ is a formula of the form $C_1 \wedge \ldots \wedge C_m$, where each C_i is a clause of the form $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$, and each $\ell_{i,j}$ is either q_k or $\neg q_k$ for some $k = 1, \ldots, m$. The task is to determine whether ϕ is satisfiable. We will build a Bayesian network $\mathcal{B}$ that corresponds to ϕ, so that: $\mathcal{B}$ will have a binary variable X, such that $P_{\mathcal{B}}(x^1) > 0$ if and only if the formula ϕ is satisfiable.

Our Bayesian network $\mathcal{B}$ will have a node Q_k for each propositional variable q_k; these variables will be roots, with $P({q_k}^1) = 0.5$. It will also have a node C_i for each clause C_i. There will be 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 will be such that it exactly duplicates the behavior of the clause.

We want to introduce a variable X to have the value 1 if and only if all the C_i's have the value 1. We can achieve this by having $C_1, \ldots, C_m$ as parents of X. This, however, requires a table of size exponential in m. We aviod this problem by introducing "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. It is easy to see that this achieves the desired effect: X has value 1 if and only if all the clauses are satisfied.

It is easy to verify that $P(x^1 \mid q_1, \ldots, q_n)$ is 1 if and only if $q_1, \ldots, q_n$ is a satisfying assignment for ϕ. As the prior probability of each possible assignment is $1/2^n$, we get that the overall probability $P(x^1)$ 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 $P(x^1) > 0$. ∎

In fact, this proof actually demonstrates a stronger result, that the problem of computing $P_{\mathcal{B}}(X)$ is in a harder complexity class: #P-hard. Similar constructions can be used to show that the MPE and MAP problems are also NP-hard.

This theorem seems like very bad news; in a way, it is. It shows that, in the worst case, the best algorithms available to date will necessarily require time that is exponential in the size of the network. In other words, assuming that P is not equal to NP, we cannot do better in the worst case than enumerating the joint. However, the worst case turns out to come up only very rarely. So, while any Bayesian network algorithm requires exponential time in the worst case, it can be quite fast in practice. As we now discuss, the same graphical structure that allows a compact representation of complex distributions also help support inference.

2 The basic ideas

We begin by considering a very simple inference task. Consider a simple chain of nodes:

$$A \rightarrow B \rightarrow C \rightarrow D$$

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

$$P(B) = \sum_A P(A)P(B \mid A)$$

Fortunately, we have all the required numbers in our Bayesian network representation: each $P(a)$ is in the CPD for A, and each $P(b \mid a)$ is in the CPD for B. Note that if A has k values and B has m values, the number of 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, i.e., k multiplications and $k-1$ additions; this process must be repeated for each of the m values b.

Now, say we want to compute $P(C)$. Using the same basic probabilistic reasoning, we have that

$$P(C) = \sum_B P(B)P(C \mid B)$$

Again, the conditional probabilities $P(c \mid b)$ are known: they constitute the CPD for C. The probability of B wasn't known, but we just computed it. Thus, we are done. Note that if C had not been conditionally independent of A given B, then the CPD for C would have included A, so that we could not have extracted the required numbers $P(C \mid B)$ directly from its CPD. Similarly, we can continue and compute $P(D)$.

It is important to note that this algorithm manipulates entire tables of numbers to produce other tables. For example, as we discussed in the $A \to B$ step, the algorithm produces an entire distribution over all of the values of B. All of these are used later in order to produce the distribution over C. This turns out to be a critical property of the general algorithm.

Now, let us analyze the complexity of this process in general. Assume that we have a chain with n nodes $X_1, \ldots, X_n$, where each node in the chain has k values. The total cost for each operation 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 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 do 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 joint probabilities. Hence, we have at least one example where, despite the exponential size of the joint distribution, we can do inference in linear time.

Thus, we have managed to do inference without generating the joint. How? Let's reexamine this process in terms of the joint. By definition, the probability $P(D)$ is simply what we get by summing out the joint:

$$\sum_{A,B,C} P(A,B,C,D)$$

By the chain rule for this Bayesian network, this is equal to

$$\sum_{A,B,C} P(A)P(B \mid A)P(C \mid B)P(D \mid C).$$

Let us open up the summation into individual summations:

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

But there is no reason to do all the multiplications first and then all the summations. Instead, we can do:

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

Note that the value of the internal summation $\sum_A P(A)P(B \mid A)$ depends only on the value of B. Thus, for each value b, we compute $f_1(b) = \sum_A P(A)P(b \mid A)$. The result, as we discussed above, is a vector of numbers $f_1(B)$, one for each value b; in this case, $f_1(B)$ corresponds exactly to $P(B)$. We can then proceed and use this distribution for computing $P(C)$, via

$$f_2(c) = \sum_B f_1(B)P(c \mid B)$$

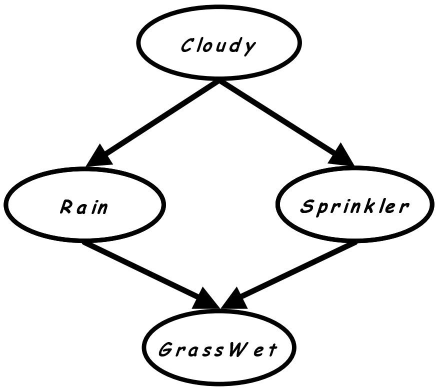

Figure 2: A simple Bayesian network with a loop.

This gives us a new vector $f_2(C)$, which we then proceed to use in later phases.

This procedure is a form of dynamic programming: doing this summation in the standard 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 bottom up instead of top down. In this case, we perform the innermost summation first, computing once and for all the values in $f_1(B)$; that allows us to compute $f_2(C)$ once and for all, etc.[1]

Let us consider a slightly more complex example, shown in Figure 2. This network represents a situation where the weather — cloudy or not — influences the chance that it will rain, as well as the gardener's decision to turn the sprinklers on. These, in turn, influence the chance that the lawn next to the building will be wet. This network has a loop in it, which, as we will see, affects this inference procedure.

Assume we want to compute $P(W)$. Again, we begin by considering the joint distribution:

$$\begin{aligned} P(W) &= \sum_{R,S,C} P(W \mid R,S)P(R \mid C)P(S \mid C)P(C) \\ &= \sum_{R,S} P(W \mid R,S) \sum_C P(R \mid C)P(S \mid C)P(C) \end{aligned}$$

Thus, to compute $P(w)$, we first compute

$$f_1(R,S) = \sum_C P(R \mid C)P(S \mid C)P(C)$$

This gives us a value $f_1(r,s)$ for every value of r,s. If both are binary, we have four different numbers. We can then use these numbers in the next phase, by computing

$$\sum_{R,S} P(W \mid R,S) f_1(R,S)$$

[1]Dynamic programming is a general principle used when the solution to a problem requires that we solve many smaller subproblems, and the same subproblem recurs many times. In this case, we are often better off precomputing the solution to the subproblems, storing them, and using them to compute the values to larger problems. Perhaps the simplest application of dynamic programming is the the problem of computing *Fibonacci numbers*, where $F_n = F_{n-1} + F_{n-2}$. The simple recursive algorithm to compute *Fibonacci(n)* recursively calls *Fibonacci(n−1)* and *Fibonacci(n−2)*. The first of these recursive calls, *Fibonacci(n − 1)*, calls *Fibonacci(n − 2)* and *Fibonacci(n − 3)*. Thus, we are already have two calls to *Fibonacci(n − 2)*. This repetition blows up, resulting in an exponential time algorithm. On the other hand, if we compute "bottom up", we start with F_1 and F_2, compute F_3 from F_1 and F_2, compute F_4 from F_2 and F_3, etc. Clearly, this process computes F_n in time linear in n. This is precisely the same principle we use here.

Procedure Variable-Elimination(
  Graph over $X_1, \ldots, X_n$, // Bayesian network structure
  $P(X_i \mid \mathbf{Pa}_{X_i})$, // CPDs for Bayesian network nodes
  $Y_1, \ldots, Y_k$ // query variables
)
  Let $\mathcal{F}$ (a set of factors) be $\{P(X_i \mid \mathbf{Pa}_{X_i}) : i = 1, \ldots, n\}$
  Let $\{Z_1, \ldots, Z_m\} = \mathbf{X} - \mathbf{Y}$
  For $i = 1, \ldots, m$
    Extract from $\mathcal{F}$ all factors $F_1, \ldots, F_r$ mentioning Z_i
    Let F' be $\prod_j F_j$
    Let F'' be $\sum_{Z_i} F'$
    Insert F'' into $\mathcal{F}$
  Return($\prod_{F \in \mathcal{F}} F$)

Figure 3: Variable Elimination algorithm without evidence

This allows us to compute the probability of the different values of w.

Again, the crucial point here is to note that our algorithm is computing, not individual probabilities, but entire tables. In this case, it's computing and maintaining entire tables $f_1(R, S)$, etc, with an entry for every value r and s.

To summarize, the two ideas that get us around the exponential blowup are:

- because of the structure of the Bayesian network, some subexpressions in the joint depend only on a small number of variables
- by computing them once and remembering the results, we can avoid generating them exponentially many times.

3 Variable Elimination

3.1 Basic elimination

Our procedure in both of these examples is an instance of a general algorithm called *Variable Elimination (VE)*, shown in Figure 3. The basic idea 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 which we enter into our set of factors to be dealt with.

Let us demonstrate the procedure on a nontrivial example. Consider the network demonstrated in Figure 4, which is a highly simplified model for lung disease diagnosis. We have that

$$P(D) = \sum_{A,B,L,T,S,X,V} P(V)P(T \mid V)P(S)P(L \mid S)P(B \mid S)P(A \mid L,T)P(X \mid A)P(D \mid A,B)$$

We will now apply the VE algorithm to compute $P(D)$, eliminating variables in the order in the summation, going from the inside out.

1. Eliminating V: We compute, for each pair of values t and v, the product $P(t \mid v)P(v)$, and the sum out to obtain
$$f_1(T) = \sum_V P(T \mid V)P(V)$$

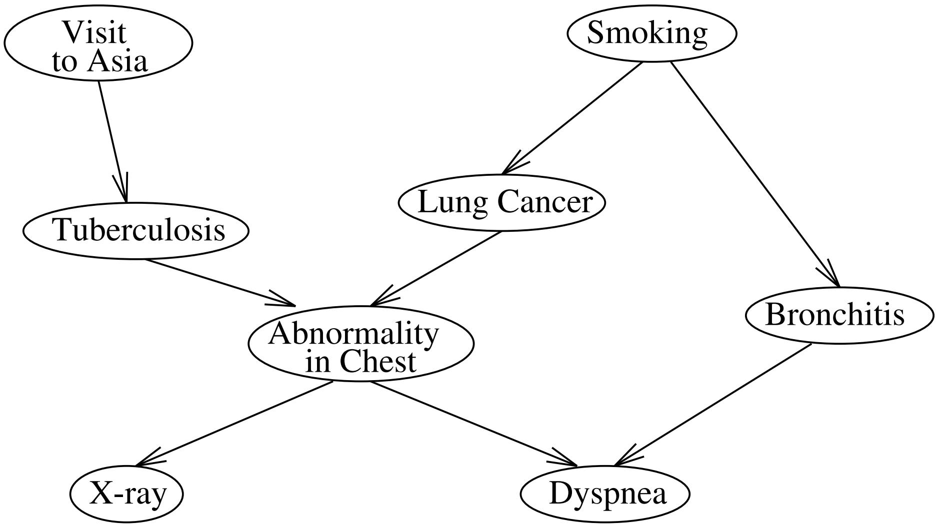

Figure 4: The Asia Bayesian network

For example, if T (Tuberculosis) has the values {no, mild, severe}, this process would end up computing a factor with three entries, one for every value of T.

2. Eliminating X: We compute

$$f_2(A) = \sum_X P(X \mid A),$$

which is a factor that has a value for each value of A.

3. Eliminating S: We compute

$$f_3(B, L) = \sum_S P(B \mid S) P(L \mid S) P(S),$$

a factor that contains a value for every pair of values b and l for the variables B and L. Thus, if Bronchitis has three values and lung cancer has four, then this would be a total of 12 values.

4. Eliminating T: Note that we have already eliminated one of the original CPDs that mention T — $P(T \mid V)$. Conversely, we introduced the factor $f_1(T)$ that mentions T. Hence, we do:

$$f_4(A, L) = \sum_T P(A \mid T, L) f_1(T)$$

5. Eliminating L:

$$f_5(A, B) = \sum_L f_3(B, L) \cdot f_4(A, L)$$

6. Eliminating B.

$$f_6(A, D) = \sum_B f_5(A, B) \cdot P(D \mid A, B)$$

7. Eliminating A:

$$f_7(D) = \sum_A f_6(A, D) f_2(A)$$

The result is the distribution $P(D)$ over the possible values of D.

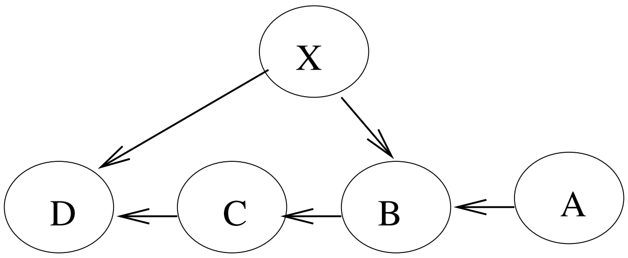

Figure 5: A Bayesian network where elimination does not lead to natural factors

We can summerize these steps in the following table:

Var eliminated	Factors used	New factor
V	$P(V)$, $P(T \mid V)$	$f_1(T)$
X	$P(X \mid A)$	$f_2(A)$
S	$P(S)$, $P(L \mid S)$, $P(B \mid S)$	$f_3(B, L)$
T	$f_1(T)$, $P(A \mid T, L)$	$f_4(A, L)$
L	$f_3(B, L)$, $f_4(A, L)$	$f_5(A, B)$
B	$f_5(A, B)$, $P(D \mid A, B)$	$f_6(A, D)$
A	$f_2(A, D)$, $f_2(A)$	$f_7(D)$

Note that we can use any elimination ordering. For example, consider eliminating variables in the order A, B, X, T, V, S, L. We would get the following factors, in order:

Var eliminated	Factors used	New factor
A	$P(A \mid L,T)$, $P(D \mid A, B)$, $P(X \mid A)$	$g_1(L,T,D,B,X)$
B	$P(B \mid S)$, $g_1(L,T,D,B,X)$	$g_2(L,T,D,X,S)$
X	$g_2(L,T,D,X,S)$	$g_3(L,T,D,S)$
T	$g_3(L,T,D,S)$, $P(T \mid V)$	$g_4(L,D,S,V)$
V	$g_4(L,D,S,V)$, $P(V)$	$g_5(L,D,S)$
S	$g_5(L,D,S)$, $P(L \mid S)$	$g_6(L,D)$
L	$g_6(L,D)$	$g_7(D)$

The result is, as before $P(D)$.

It is interesting to consider the semantics of the intermediate factors generated as part of this computation. Often, they correspond to marginal or conditional probabilities. For example, with a little thought we can convince ourselves that the factor g_1 in this last computation represents $P(X, D, A \mid T, L, B)$. However, while these factors often correspond to such probabilities, this is not always the case. Consider, for example, the network of Figure 5. The result of eliminating the variable A is a factor that does not correspond to any probability or conditional probability. To see this, we note that this term cannot correspond to a probability where A or C are on the left hand side of the conditioning bar, because we have not yet multiplied in their CPDs. Therefore, one might think that this factor corresponds to $P(D, B \mid A, C)$, but that would also not be the case, because we cannot condition B on C without the CPD $P(C \mid B)$.

3.2 Dealing with evidence

It remains only to consider how we would introduce evidence. For example, assume we observe s_1 (the person smokes) and x_0 (the X-ray came out negative). First, we reduce this problem to computing $P(D, s_1, x_0)$, i.e., $P(d, s_1, x_0)$ for every value d of D. How can we use that? As usual:

$$P(d \mid s_1, x_0) = \frac{P(d, s_1, x_0)}{P(s_1, x_0)} = \frac{P(d, s_1, x_0)}{\sum_d P(d, s_1, x_0)}$$

In other words, once we compute a factor over D that represents the joint probability $P(D, s_1, x_0)$, we simply renormalize it to get the desired conditional probability.

How do we compute $P(D, s_1, x_0)$? Let us go back to the joint distribution. To compute any individual value $P(d, s_1, x_0)$, we want to sum out only those entries in the joint where the values D, S, X take on these particular values. In other words:

$$\begin{aligned} P(d, s_1, x_0) &= \textstyle\sum_{A,B,L,T,V} \; P(V)P(T \mid V)P(s_1)P(L \mid s_1) \\ &\qquad P(B \mid s_1)P(A \mid L, T)P(x_0 \mid A)P(d \mid A, B) \end{aligned}$$

We can execute this computation in exactly the same way, by eliminating the unnecessary variables — all except D, S, X — in the usual way. We do not eliminate D because we are interested in getting a factor over its values; we do not eliminate S and X because their values are fixed.

Let us use the same elimination ordering that we used in our first computation before (the one that generated the f factors rather than the g factors):

1. Eliminating V: This phase is unchanged; it generates the same $f_1(T)$.
2. Eliminating X: This phase now involves no operations.
3. Eliminating S: This phase also involves no operations.
4. Eliminating T: This phase is unchanged; it generates the same $f_4(A, L)$.
5. Eliminating L: This phase is now different. Note that, before, the summation over S in phase (3) caused us to sum out various factors into a single factor $f_3(B, L)$. In this computation, this did not happen. In our case, the summation looks like:

   $$f_5'(A) = \sum_L f_4(A, L)P(L \mid s_1)$$

   Note that this term doesn't depend on S, because only one value of S is involved; i.e., $P(L \mid s_1)$ has exactly one number for every value of L. Also note that f_5', unlike f_5, does not depend on B. Intuitively, observing S decorrelates B and L, rendering them conditionally independent. From the algorithm's perspective, we never compute $f_3(B, L)$, which forced us to introduce the dependence on B.
6. Eliminating B:

   $$f_6'(A) = \sum_B P(B \mid s_1)P(D \mid A, B)$$

7. Eliminating A:

   $$f_7'(D) = \sum_A f_6'(A)P(x_0 \mid A)$$

8. The result: To get $P(D, s_1, x_0)$, the result needs to be multiplied by the remaining term $P(s_1)$. (Note that, for getting the conditional $P(D \mid s_1, x_0)$, this last multiplication by a constant factor is irrelevant.)

We can view inference with evidence as executing an identical process to our original inference algorithm in Figure 3. The only difference is that we first *reduced* our factors to eliminate the parts that are inconsistent with our evidence. More precisely:

Procedure Variable-Elimination(
  Graph over $X_1, \ldots, X_n$, // Bayesian network structure
  $P(X_i \mid \mathbf{Pa}_{X_i})$, // CPDs for Bayesian network nodes
  $u_1, \ldots, u_m$, // evidence
  $Y_1, \ldots, Y_k$ // query variables
)
  Let $\mathcal{F}$ (a set of factors) be $\{P(X_i \mid \mathbf{Pa}_{X_i})\}_{i=1}^{n}$
  For each $g \in \mathcal{F}$
    replace g by $g|_{\mathbf{U}=\mathbf{u}}$
  Let $\{Z_1, \ldots, Z_m\} = \mathbf{X} - \mathbf{Y} - \mathbf{U}$
  For $i = 1, \ldots, m$
    Extract from $\mathcal{F}$ all factors $F_1, \ldots, F_r$ mentioning Z_i
    Let F' be $\prod_j F_j$
    Let F'' be $\sum_{Z_i} F'$
    Insert F'' into $\mathcal{F}$
  Return($\prod_{F \in \mathcal{F}} F$)

Figure 6: Variable Elimination algorithm with evidence

Definition 3.1: Let $g(X_1, \ldots, X_k)$ be a factor, U a variable, and $u \in Val(U)$. We define $g|_{U=u}$ to be as follows. If $U \notin \{X_1, \ldots, X_k\}$, then $g|_{U=u} = g$. Otherwise, if $U = X_i$, then $g|_{U=u}$ is a factor over $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_k$ such that

$$g|_{Y=y}(x_1, \ldots, x_{i-1}, u, x_{i+1}, \ldots, x_k).$$

We generalize this notion to a set $g|_{\mathbf{U}=\mathbf{u}}$ in the obvious way. ∎

If we first reduce all factors to that parts that are consistent with our evidence, and then sum out the remaining, the result is equivalent to summing out only the entries in the joint distribution that are consistent with the evidence. The resulting algorithm is shown in Figure 6.

4 Complexity and Graph Structure

Let's do a very rough analysis of the complexity of this algorithm. For simplicity, we assume that all variables have the same number of values, say v.

At each stage, the algorithm picks a variable, then multiplies all factors involving that variable. The result is a single large factor. The variable then gets summed out of this factor, leading to a somewhat smaller factor. Thus, the work revolves around these factors that get created and processed.

How much work is it to create a factor f? Let N be the number of entries in f. To create a single entry in f, we multiply some number of other factors, at most $O(n)$. Therefore, we do at most a linear amount of work (linear in the number of nodes in the network) for each entry in f. When we sum out f, each entry gets touched at most once. Therefore, we do a constant amount of work for the summation. Overall, the amount of work is $O(n)$ for each entry in f. As the number of factors is also linear in n, it is clear that any exponential blowup must arise from the size of the intermediate factors generated. Indeed, the number of entries in a factor containing m variables is v^m. To summarize, we see that the computational cost of the VE algorithm is dominated by the sizes of the intermediate factors generated.

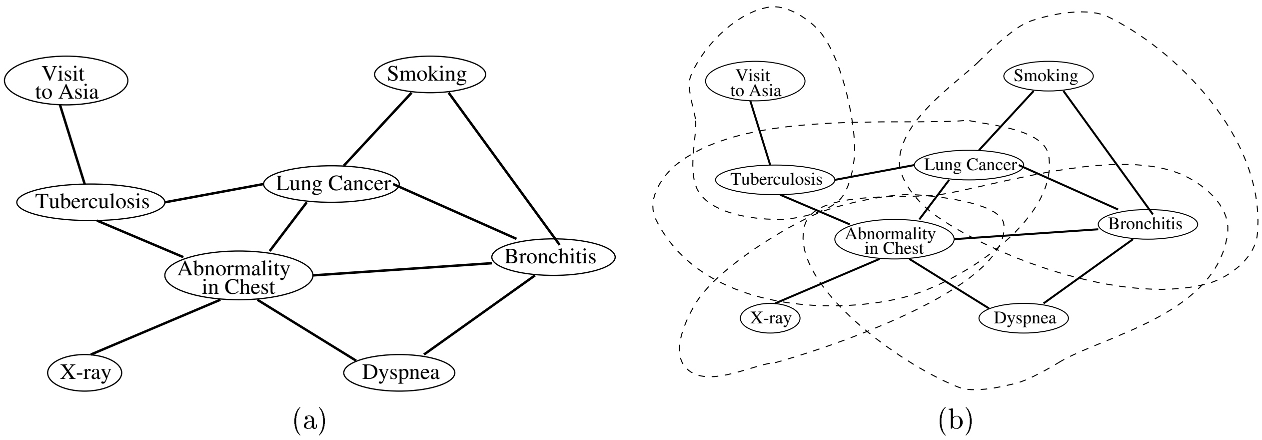

Figure 7: (a) Induced graph for Asia network. (b) Cliques in induced graph.

4.1 Induced graphs

To analyze this quantity in terms of the graph, we will construct a useful data structure called the *induced graph*.

Definition 4.1: Let G be a Bayesian network structure over $X_1, \ldots, X_n$, and $Y_1, \ldots, Y_k$ be an elimination ordering for G. The *induced graph* H is an undirected graph over $X_1, \ldots, X_n$, where X_i and X_j are connected by an edge if they both appear in some intermediate factor f generated by the VE algorithm eliminating $Y_1, \ldots, Y_k$ in this order. ∎

Let's construct the induced graph for our Asia example. Let us first introduce the edges that must be present regardless of the elimination ordering. In particular, each CPD must participate in some expression. A CPD induces the obvious edges: an edge between each node and its parents. It also introduces other edges: an edge between any pair of parents of a given node. This graph is the moralized graph that we encountered in the context of Markov networks. Thus, as a Markov network, it is the minimal I-map for the Bayesian network.

Now, let us consider the factors that were constructed in the different steps of the summation are:

$$
\begin{array}{lcc}
(1) & V & \{V, T\} \\
(2) & X & \{X, A\} \\
(3) & S & \{S, L, B\} \\
(4) & T & \{A, L, T\} \\
(5) & L & \{A, L, B\} \\
(6) & A & \{A, D, B\} \\
(7) & B & \{D, B\}
\end{array}
$$

Neither the first step nor the second introduce any new edges. The third step introduces an edge between B and L, because they participate in computing $f_3(B, L)$. The remaining steps introduce no other edges. The resulting induced graph is shown in Figure 7(a).

The induced graph for an ordering reveals the computational cost of the associated variable elimination algorithm. For an undirected graph H, we define a *complete subgraph* of H to be a subset of the vertices of H where each vertex is connected by an edge to all other vertices in the subgraph. We define a *clique* to be a maximal complete subgraph, i.e., one to which no additional vertices can be added.

We can now show the following theorem:

Theorem 4.2: *Let H be the induced graph for some ordering, and consider the variable elimination computation for that ordering.*

1. *Every clique in the induced graph corresponds to an intermediate factor in the computation.*
2. *Every factor generated during the variable elimination process is a subset of some maximal clique in the graph.*

Before proving this theorem, let's check that these properties hold for our graph. As we can see in Figure 7(b), our maximal complete subgraphs are

$$\begin{aligned} C_1 &= \{V, T\} \\ C_2 &= \{X, A\} \\ C_3 &= \{S, L, B\} \\ C_4 &= \{A, L, T\} \\ C_5 &= \{A, L, B\} \\ C_6 &= \{A, B, D\} \end{aligned}$$

It is easy to verify that both these properties hold for this set of cliques.

Proof: The second statement is easier, so we prove it first. Consider a factor $f(Y_1, \ldots, Y_k)$ 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 $Y_1, \ldots, Y_k$ are a complete subgraph; either this is a maximal complete subgraph (a clique) or a subset of a larger one.

To prove the first statement, consider some clique $Y_1, \ldots, Y_k$. Assume, without loss of generality, that Y_1 is the first variable among this set to be eliminated. As this 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, these edges 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, we get a factor 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. ∎

Thus, there is a direct correspondence between the maximal factors generated by our algorithm and cliques in the induced graph. Note, however, that the induced graph depends on the elimination ordering. Consider, for example, our other elimination ordering for the Asia network. In this case, we can verify that our induced graph has all possible edges with the exception of: S—A, V—A, V—X, and V—B. The largest clique contains six nodes — L, T, A, D, B, X. In our original elimination ordering, the largest clique contained only three nodes. Therefore, the cost of computation here is substantially more expensive.

This observation immediately raises the question of finding the best elimination ordering — the one where the largest clique in the induced graph is smallest. Unfortunately, it turns out that the problem of finding a good elimination ordering is also NP-hard. More precisely, it is NP-hard to decide, given a graph and some bound K, whether there is an elimination ordering where the largest clique in the induced subgraph is of size at most K. This NP-hardness result is distinct from the NP-hardness of inference itself. That is, even if some oracle gives us the best elimination ordering, the inference task using that ordering can still require exponential time.

Thus, we cannot easily tell by looking at a graph how computationally expensive inference on it will be. However, as usual, NP-hardness is not the end of the story. First, it turns out that there

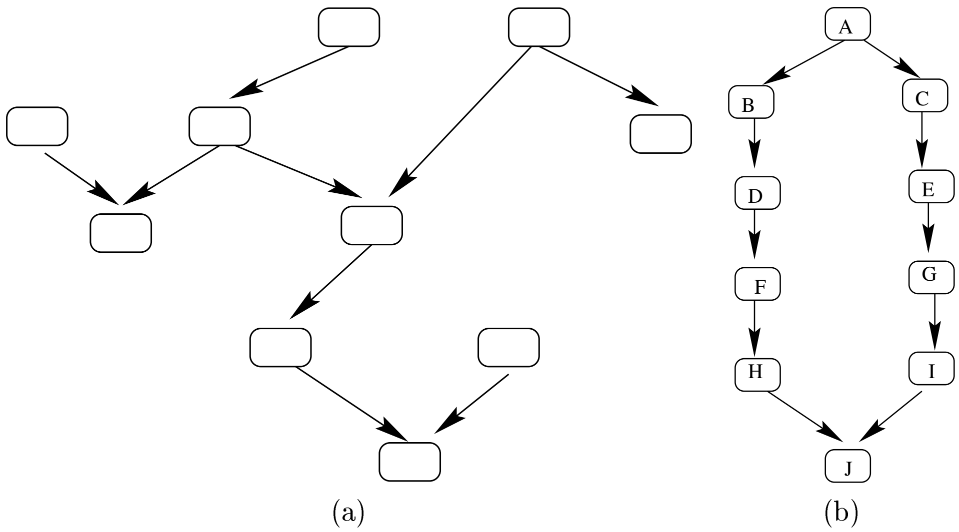

Figure 8: Examples of a polytree and a non-polytree.

is an algorithm that can find the optimal ordering in time which is exponential in the size of the largest clique; since the inference process takes that long anyway, we may be willing to spend the time *a priori* in order to make our computation more efficient. If we are unable to spend that time, there are several reasonable heuristics that find orderings that are not too bad in practice. Two that are most used in practice are:

- **Maximum cardinality search:** Number the nodes from 1 to n, in increasing order, always assigning the next number to the vertex having the largest set of previously numbered neighbors. The elimination order then proceeds from n to 1.
- **Minimum discrepancy search:** At each point, eliminate the node that causes the fewest edges to be added to the induced graph.
- **Minimum size/weight search:** At each point, eliminate the node that causes the smallest clique to be created, where "small" is measured either in terms of number of nodes or number of entries in the factor. (If all nodes have the same number of values, these two measures are identical.)

Most Bayesian network systems use one of these heuristics.

Finally, there are classes of graphs where we can determine, in advance, that a good elimination ordering exists. One very obvious class which has received a lot of attention is the class of polytrees, or singly connected Bayesian networks.

Definition 4.3: A Bayesian network is a *polytree* if it is singly connected, i.e., if in the underlying undirected graph, there is at most one path between any pair of nodes. ∎

Thus, for example, the graph in Figure 8(a) is a polytree, whereas the graph in Figure 8(b) is not.

For polytrees, it is easy to see that we can do inference efficiently.

Theorem 4.4: *For the class of polytree Bayesian networks, the problem of computing $P(X, \mathbf{y})$ for any X in the network can be accomplished in linear time in the size of the network.*

Proof: An elimination ordering that works from the leaves inwards induces a graph which is precisely the moral graph. Therefore, the largest clique is precisely the largest family in the graph. Assuming that CPDs are represented as tables, this quantity is linear in the size of the network (which is measured as the total size of its description, including the CPDs). ∎

Polytrees received a lot of attention in the early days of Bayesian networks because the first commonly-known inference algorithm for any type of Bayesian network was Pearl's message passing algorithm for polytrees (see Exercise **??**). In fact, for a long time there was widely held misconception that the line between polytrees and other networks was a sharp cutoff, so that everything that is not a polytree is immediately hard. The analysis that we showed here shows that this is not the case; rather, there is a continuum of complexity defined by the size of the largest clique in the graph. While we cannot immediately estimate this cost, in practice, the number of distinct paths between parts of the graph is a good way of getting a rough estimate for how expensive inference in a given network is likely to be. We return to this issue in the next chapter.