4

Exact Inference

We now turn our attention to the problem of *inference* in graphical models (Figure 4.1). Generally speaking, inference is the process of drawing conclusions from available evidence. Inference can also be framed as a "reasoned change in view" (Harman, 1986) or "reasoning with beliefs" (Boghossian, 2014). Through this lens, inference can be a dynamic process where beliefs are iteratively challenged and updated as new evidence is introduced.

Given a probabilistic model (such as a Bayesian network or MRF), we perform inference to answer useful questions about the world. Our model can help us answer questions like:

- *What is the posterior distribution over a variable of interest?*
- *What is the most probable explanation for a given hypothesis?*
- *How certain are we that this explanation holds?*

In this chapter, we present means of answering such questions. We will mainly consider the problem of computing posterior probabilities of the form $p(\mathbf{y} \mid \mathbf{x})$ for *query variables* $\mathbf{y}$ given *observed evidence* $\mathbf{x} = (x_1, \ldots, x_n)$ (Zhang and Poole, 1996). Formally, we will focus on two types of queries:

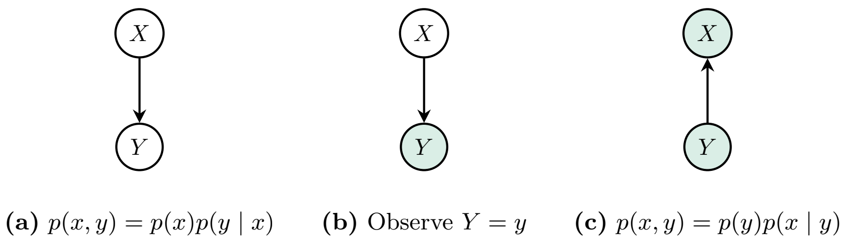

X
Y
(a) $p(x,y) = p(x)p(y \mid x)$
X
Y
(b) Observe $Y = y$
X
Y
(c) $p(x,y) = p(y)p(x \mid y)$

Figure 4.1: A simple case of inference over a graphical representation of Bayes' rule (figure and caption adapted from Bishop 2006, Chapter 8.4). Let $p(x,y)$ be a joint distribution over variables X and Y. We can factorize this joint as $p(x)p(y \mid x)$ by applying the chain rule (Definition 2.28), as graphically represented by **(a)**. Next, we observe the value taken by Y (denoted by shading) and treat $p(x)$ as a prior over the latent variable X **(b)**. We now wish to infer the posterior distribution $p(x \mid y)$. By the sum and product rules, $p(y) = \sum_{x'} p(y \mid x')p(x')$. By Bayes' rule (Definition 2.29), $p(x \mid y) = p(y \mid x)p(x)/p(y)$. Thus, we can rearrange the joint factorization as $p(y)p(x \mid y)$, as represented by **(c)**.

1. *Marginal inference.* What is the probability of a given variable in our model after we sum everything else out, e.g.,

$$p(y=1) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} p(y=1, x_1, x_2, \ldots, x_n).$$

2. *Maximum a posteriori (MAP) inference.* What is the most likely assignment to the variables in the model, e.g.,

$$\hat{\mathbf{x}} = \operatorname*{argmax}_{x_1,\ldots,x_n} p(y=1, x_1, \ldots, x_n).$$

In this chapter, we place our initial focus on *exact* probabilistic inference in graphical models. This is a challenging task that is NP-hard in the general case (Cooper, 1990). Nevertheless, tractable solutions can be obtained for certain kinds of problems. As illustrated throughout this chapter, the tractability of an inference problem depends heavily on the structure of the graph that describes the probability of interest.

To begin, we will introduce three fundamental and inter-related algorithms for exact inference: variable elimination (Section 4.1), belief propagation for tree-structured graphs (Section 4.2), and the junction tree algorithm (Section 4.3). We will then discuss the computational complexity of MAP versus marginal inference and present a couple of algorithms for efficient MAP inference (Section 4.4). In Chapter 5, we will continue our discussion in the context of *approximate inference* for the setting where exact inference is intractable.

4.1 Variable Elimination

We first introduce *variable elimination*, a fundamental exact inference method for answering queries to MRFs and Bayesian networks (Zhang and Poole, 1994). Variable elimination is a general approach that can be used for both marginal and MAP inference. It efficiently computes quantities of interest by serially *eliminating* variables that are irrelevant to the query. Given a model over variables $\mathbf{X}$, with joint distribution $p(\mathbf{x})$, variable elimination can be used to answer queries such as conditional probabilities — for example, computing a distribution of the form

$$p(\mathbf{x}_q \mid \mathbf{x}_e),$$

where $\mathbf{X}_q$ is a set of query variables, $\mathbf{X}_e$ is a (potentially empty) set of evidence variables, and the remaining variables in $\mathbf{X}$ are marginalized out. Note that for MAP inference, this query will take a different form. At a high-level, variable elimination exploits two convenient observations (Koller and Friedman, 2009):

1. By the factorization of Bayesian networks (Definition 3.2), factors often depend on a relatively small subset of variables in the joint. We can reduce the number of parameters in our model by leveraging this fact (see Section 3.1.1).

2. With dynamic programming, we can compute subexpressions in the joint and memoize to prevent exponential increases in computation.

This procedure repeatedly performs two factor operations: *product* and *marginalization*. Conveniently, we can interchange product and sum operations to reduce the number of computations. For example, consider three factors: ϕ_x with scope $\mathbf{X}$, and ϕ_y with scope $\mathbf{Y}$, and ϕ_{yz} with scope $\mathbf{Y} \cup \mathbf{Z}$. Because $\mathbf{Y}$ is in scope for ϕ_y and ϕ_{yz} only, we can interchange

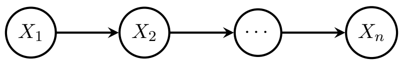


Figure 4.2: A chain-structured Bayesian network with n variables.

sum and product as follows:

$$\sum_{\mathbf{y}} \phi_x \phi_y \phi_{yz} = \phi_x \sum_{\mathbf{y}} \phi_y \phi_{yz}.$$

This simple rearrangement, whereby we *push* the summation deeper into the product, is the crux of variable elimination.

For ease of exposition, we will assume for the remainder of this chapter that all random variables are discrete variables taking k possible values each. While the principles behind variable elimination also extend to many continuous distributions (e.g., Gaussians), we will not discuss these extensions here.

We present the full variable elimination procedure in Algorithm 1. To lay the groundwork, we first describe the special case of marginal inference in chain-structured graphs. Building on this, we then present the general case, known as the *sum-product variable elimination algorithm*, and describe how it can be used for both marginal and MAP inference queries, as well as queries that condition on observed evidence.

4.1.1 Marginal Inference in a Chain-Structured Graph

We will first use a simple chain-structured graph as an illustrative example for several useful concepts.

Suppose for simplicity that we are given a Bayesian network that resembles Figure 4.2. This network represents a joint probability distribution of the form

$$p(x_1, \ldots, x_n) = p(x_1) \prod_{i=2}^{n} p(x_i \mid x_{i-1}).$$

Assume that we are interested in computing the marginal probability $p(x_n)$. The naive way of calculating this is to sum the probability over all k^{n-1} assignments to $x_1, \ldots, x_{n-1}$:

$$p(x_n) = \sum_{x_1} \cdots \sum_{x_{n-1}} p(x_1, \ldots, x_n).$$

However, we can do much better by leveraging the factorization of our joint probability distribution. This factorization allows us to rewrite the sum in a way that pushes certain variables deeper into the product.

$$\begin{aligned} p(x_n) &= \sum_{x_1} \cdots \sum_{x_{n-1}} p(x_1) \prod_{i=2}^{n} p(x_i \mid x_{i-1}) \\ &= \sum_{x_{n-1}} p(x_n \mid x_{n-1}) \sum_{x_{n-2}} p(x_{n-1} \mid x_{n-2}) \cdots \sum_{x_1} p(x_2 \mid x_1) p(x_1). \end{aligned}$$

We sum the inner terms first, starting from x_1 and ending with x_{n-1}. Concretely, we start by computing an intermediary factor

$$\tau(x_2) = \sum_{x_1} p(x_2 \mid x_1) p(x_1)$$

by summing out x_1. This takes $O(k^2)$ time because we must sum over x_1 for each assignment to x_2. The resulting factor $\tau(x_2)$ can be thought of as a table of k values (though not necessarily probabilities), with one entry for each assignment to x_2 (just as factor $p(x_1)$ can be represented as a table). We can then rewrite the marginal probability using τ as

$$p(x_n) = \sum_{x_{n-1}} p(x_n \mid x_{n-1}) \sum_{x_{n-2}} p(x_{n-1} \mid x_{n-2}) \cdots \sum_{x_2} p(x_3 \mid x_2) \tau(x_2).$$

Note that this has the same form as the initial expression, except that we are summing over one fewer variable. Thus, we are sequentially *eliminating* one variable at a time, which gives the algorithm its name.

We can therefore compute another factor

$$\tau(x_3) = \sum_{x_2} p(x_3 \mid x_2) \tau(x_2),$$

and repeat the process until we are only left with x_n, which yields a table of probabilities representing $p(x_n)$. Since each step takes $O(k^2)$ time, and we perform $O(n)$ steps, inference now takes $O(nk^2)$ time, which is much better than our naive $O(k^n)$ solution.

4.1.2 The Sum-Product Variable Elimination Algorithm

While the previous example illustrated marginal inference in a chain-structured graph, we now present variable elimination as a general procedure applicable to arbitrary graphical models with factorized representations — including both Bayesian networks and MRFs. Later, we show how, in addition to marginal inference, this procedure can be used for MAP inference and computing conditional distributions.

We first focus here on the *sum-product formulation* of variable elimination. This procedure repeatedly performs two factor operations: *product* and *marginalization*. Given two factors ϕ_1 and ϕ_2, the factor product operation simply defines the product $\phi_3 := \phi_1 \times \phi_2$ as

$$\phi_3(\mathbf{x}) = \phi_1(\mathbf{x}^{(1)}) \times \phi_2(\mathbf{x}^{(2)}).$$

The scope of ϕ_3 is defined as the union of the variables in the scopes of ϕ_1 and ϕ_2. We use $\mathbf{x}^{(i)}$ to denote an assignment to the variables in the scope of ϕ_i defined by the restriction of $\mathbf{x}$ to that scope. For example, we define $\phi_3(a, b, c) := \phi_1(a, b) \times \phi_2(b, c)$.

Next, the marginalization operation "locally" eliminates a set of variables from a factor. If we have a factor $\phi(\mathbf{x}, \mathbf{y})$ over two sets of variables $\mathbf{X}$ and $\mathbf{Y}$, marginalizing $\mathbf{Y}$ produces a new factor

$$\tau(\mathbf{x}) = \sum_{\mathbf{y}} \phi(\mathbf{x}, \mathbf{y}),$$

where the sum is over all joint assignments to the set of variables $\mathbf{Y}$. We use τ to refer to the marginalized factor. It is important to understand that this factor does not necessarily correspond to a probability distribution, even if ϕ was a conditional probability distribution.

Given the definitions of the product and marginalization operations over factors, the sum-product variable elimination algorithm proceeds as follows. Suppose we are given a graphical model $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ with an associated set of d factors $\Phi := \{\phi_i\}_{i=1}^{d}$, and we wish to compute the marginal distribution over a subset of query variables $\mathbf{X}_q \subseteq \mathbf{V}$, i.e., we aim to compute $p(\mathbf{x}_q)$. To do this, we must eliminate the remaining variables $\mathbf{X}_m := \mathbf{V} \setminus \mathbf{X}_q$ by summing them out. We begin by selecting an *elimination ordering* over the variables in $\mathbf{X}_m$ (see Section 4.1.4); for instance, in a Bayesian network, a topological ordering is one possible choice. Then, for each variable in the ordering, we identify the subset of factors in Φ that mention the variable, and on these we perform the product factor operation and then the marginalization factor operation

Algorithm 1 *Sum-Product Variable Elimination*, adapted from Zhang and Poole (1996) and Koller and Friedman (2009)

Input:

- $\Phi := \{\phi_i\}_{i=1}^d$: Set of factors implied by graph $\mathcal{G} = (\mathbf{V}, \mathbf{E})$
- $\mathbf{X}_q \in \mathbf{V}$: Set of query variables
- $\mathcal{O}$: Elimination ordering for $\mathbf{V} \setminus \mathbf{X}_q$

Output: $\phi^* = p(\mathbf{x}_q)$

```
1: for $X \in \mathcal{O}$, following the elimination ordering do
2:     Remove variable $X$ from $\mathcal{O}$
3:     Remove from $\Phi$ all $\{\phi_j\}_{j=1}^k$ where $X \in \mathit{Scope}[\phi_j]$
4:     Add new factor $\sum_x \prod_{j=1}^k \phi_j$ to $\Phi$
5: $\phi^* \leftarrow \prod_{i=1}^d \phi_i$
6: return $\phi^*$
```

(which sums out the given variable). The result is a new factor that replaces this subset of factors in Φ, and the process continues. This sequentially reduces the number of factors and eliminates variables one by one. Conceptually, the algorithm pushes summations as far inside the product of factors as possible, as illustrated in the chain-structured example. At the end of the procedure, a single factor remains, defined over the query variables $\mathbf{X}_q$, which represents the unnormalized marginal distribution $p(\mathbf{x}_q)$. The full pseudocode for the sum-product variable elimination algorithm is provided in Algorithm 1.

Two Illustrative Examples

To make Algorithm 1 concrete, we revisit the chain-structured example from Figure 4.2 to illustrate how these steps play out. In this case, the chosen ordering was simply the topological ordering $x_1, x_2, \ldots, x_{n-1}$. Starting with x_1, we collected all the factors involving x_1, which were $p(x_1)$ and $p(x_2 \mid x_1)$. We then used them to construct a new factor

$$\tau(x_2) = \sum_{x_1} p(x_2 \mid x_1) p(x_1).$$

This can be seen as the results of Steps 1 and 2 of variable elimination in Algorithm 1: first we form a large factor $p(x_2, x_1) = p(x_2 \mid x_1)p(x_1)$, then we eliminate x_1 from that factor to produce τ. Next, we repeat the same procedure for x_2, except that the factors are now $p(x_3 \mid x_2), \tau(x_2)$.

As a second illustrative example of the variable elimination algorithm, we walk through a slightly more complex application of Algorithm 1. Recall the graphical model of student performance that we introduced earlier (Figure 3.1). The probability specified by the model is of the form

$$p(l, g, i, d, s) = p(l \mid g)p(s \mid i)p(i)p(g \mid i, d)p(d).$$

Let's suppose that we are computing $p(l)$ and are eliminating variables in their topological ordering in the graph. First, we eliminate d, which corresponds to creating a new factor

$$\tau_1(g, i) = \sum_d p(g \mid i, d)p(d).$$

Next, we eliminate i to produce a factor

$$\tau_2(g, s) = \sum_i \tau_1(g, i)p(i)p(s \mid i).$$

Then, we eliminate s yielding $\tau_3(g) = \sum_s \tau_2(g, s)$, and so on. Note that these operations are equivalent to summing out the factored probability distribution as follows:

$$p(l) = \sum_g p(l \mid g) \sum_s \sum_i p(s \mid i)p(i) \sum_d p(g \mid i, d)p(d).$$

This example requires computing at most k^3 operations per step: each factor is at most over 2 variables, and one variable is summed out at each step (the dimensionality k in this example is either 2 or 3).

Computational Complexity

Though variable elimination is often faster than the naive approach of inference by enumeration, it is still NP-hard. In short, the computational complexity of variable elimination is exponential in the size of the largest factor. However, it is very important to understand that the running time of variable elimination depends heavily on the structure of the graph. More concretely, the running time depends on the following structural properties of the Bayesian network or MRF $\mathcal{G}$.

- *Total variables in $\mathcal{G}$.* Let $n = |\mathbf{V}|$ denote the number of variables in the system.

- *Treewidth.* Treewidth w, which measures how close the graph is to a tree (Definition 2.43). In this setting, the treewidth can be shown to be equivalent to the maximum size of any factor τ formed during the variable elimination algorithm.

- *Domain size of variables in* $\mathbf{V}$. Let k_i denote the size of the domain (i.e., the number of possible values) for each $V_i \in \mathbf{V}$. For ease of exposition, we can assume that all variables have k states.

Thus, we can express the time complexity of variable elimination as $O(nk^{w+1})$. As $w = 1$ in trees (including chains), the computational complexity of variable elimination on tree-structured graphs is linear in n: $O(nk^2)$. For graphs with large treewidth, variable elimination can become intractable for large n.

4.1.3 Introducing Evidence

Above we have described the variable elimination algorithm for the case of marginal inference. A closely related and equally important problem is the computation of conditional probabilities of the form

$$p(\mathbf{x}_q \mid \mathbf{x}_e = \mathbf{e}) = \frac{p(\mathbf{x}_q, \mathbf{x}_e = \mathbf{e})}{p(\mathbf{x}_e = \mathbf{e})},$$

where $p(\mathbf{x}_q, \mathbf{x}_e, \mathbf{x}_m)$ is a probability distribution over *query* variables $\mathbf{X}_q$, *observed evidence* variables $\mathbf{X}_e$, and *unobserved* variables $\mathbf{X}_m$.

We can compute this probability by performing variable elimination once on $p(\mathbf{x}_q, \mathbf{x}_e = \mathbf{e})$ and then once more on $p(\mathbf{x}_e = \mathbf{e})$. To compute $p(\mathbf{x}_q, \mathbf{x}_e = \mathbf{e})$, we simply take every factor $\phi(\mathbf{x}'_m, \mathbf{x}'_q, \mathbf{x}'_e)$ which has scope over variables $\mathbf{X}'_e \subseteq \mathbf{X}_e$, and we set their values to $\mathbf{e}$. Then we perform standard variable elimination to obtain a factor over $\mathbf{X}_q$ only.

4.1.4 Elimination Orderings

The variable elimination algorithm requires an *ordering* over the variables, according to which variables will be eliminated. In our chain example, we simply took the ordering implied by the DAG. However, it is important to note that the time complexity of variable elimination is sensitive to the order in which variables are eliminated. Further, it is NP-hard to find the optimal ordering.

In practice, we can resort to the following heuristics to obtain an ordering for variable elimination. These methods often result in reasonably good performance in many interesting settings (Koller and Friedman, 2009).

- *Min-neighbors*, where we choose the next variable with the fewest dependent variables.
- *Min-weight*, where we choose variables to minimize the product of the cardinalities of dependent variables.
- *Min-fill*, where we choose vertices to minimize the size of the factor that will be added to the graph.

4.1.5 Generalizing Factor Operations

We have so far described the sum-product formulation of variable elimination for marginal (and conditional) inference, involving the *product* and *marginalization* factor operators. We now generalize these operations, enabling alternative inference procedures with the variable elimination algorithm, including MAP inference.

The premise of variable elimination rests on the properties of *commutative semirings*: a set **S** and two binary operations that are associative, commutative, and satisfy identity laws. Commutative semirings satisfy the distributive law, whereby

$$(x \times y) + (x \times z) = x \times (y + z) \tag{4.1}$$

for all $x, y, z \in \mathbf{S}$. Note that the distributive law reduced the number of computations in Equation 4.1 from three (on the lefthand side) to two (on the righthand side). See Golan (2013) for a formal discussion of commutative semirings.

The associative, commutative, and distributive properties of these binary operations allow us to perform more efficient inference. When the

	Domain	+	×
Sum-product	$[0, \infty)$	$(+, 0)$	$(\times, 1)$
Max-product	$[0, \infty)$	$(\max, 0)$	$(\times, 1)$
Min-sum	$(-\infty, \infty]$	$(\min, \infty)$	$(+, 0)$
Boolean satisfiability	$\{T, F\}$	$(\vee, F)$	$(\wedge, T)$

Table 4.1: Examples of common commutative semirings. Table from Murphy (2012).

binary operations of a commutative semiring are taken to be addition and multiplication, we obtain the sum-product formulation of variable elimination:

$$p(\mathbf{x}_q \mid \mathbf{x}_e) \propto \sum_{\mathbf{x}} \prod_{c \in \mathbf{C}} \phi_c(\mathbf{x}_c),$$

where $\mathbf{X}_q$ is a set of query variables, $\mathbf{C}$ is a set of cliques c, and $\mathbf{X}_e$ is a set of evidence variables whose values are fixed rather than marginalized over (Murphy, 2012). Notably, replacing summation with the *max* operation in variable elimination yields a version of the algorithm for MAP inference. See Table 4.1 for additional commutative semirings.

4.1.6 Limiting Extraneous Computation

We have seen how the variable elimination algorithm can answer marginal queries of the form $p(\mathbf{x}_q \mid \mathbf{x}_e = \mathbf{e})$ for both directed and undirected networks. However, this algorithm has an important shortcoming: if we want to ask the model for another query (e.g., $p(\mathbf{x}'_q \mid \mathbf{x}'_e = \mathbf{e}')$), we need to restart the algorithm from scratch. This is very wasteful and computationally burdensome.

Fortunately, it turns out that this problem is also easily avoidable. When computing marginals, variable elimination produces many intermediate factors τ as a side-product of the main computation. These factors turn out to be the same as the ones that we need to answer other marginal queries. By caching them after a first run of variable elimination, we can easily answer new marginal queries at essentially no additional cost.

4.2 Belief Propagation in Trees

We now introduce *belief propagation*: a family of exact inference algorithms based on recursive *message passing*. Belief propagation was first introduced by Judea Pearl (Pearl, 1982; Pearl, 1986), but the more general concept of message passing takes a wide range of important algorithms as special cases: the forward/backward algorithm, the Viterbi algorithm, turbo decoding, the Kalman filter, and more (Kschischang *et al.*, 2001; Yedidia *et al.*, 2003; McEliece *et al.*, 1998). Belief propagation performs exact inference over tree-structured graphs, both undirected and directed (Definitions 2.40, 2.41). In Section 4.3, we will introduce the junction tree algorithm: an extension of belief propagation that works for general networks by transforming the input graph into a tree whose nodes represent cliques.

Belief propagation first executes two runs of the variable elimination algorithm to initialize a data structure that holds a set of pre-computed factors. Once the structure is initialized, it can answer marginal queries in $O(1)$ time. You can think of variable elimination and belief propagation as two flavors of the same technique: top-down dynamic programming versus bottom-up table filling, respectively. Just like in computing the n^{th} Fibonacci number F_n, top-down dynamic programming (i.e., variable elimination) computes *just* that number, but bottom-up (i.e., belief propagation) will create a filled table of all F_i for $i \leq n$. Moreover, the two-pass nature of belief propagation is a result of the underlying dynamic programming on bi-directional trees, while the Fibonacci numbers' relation is a uni-directional tree.

In parallel with the sum-product and max-product variants of variable elimination, we now introduce two corresponding forms of belief propagation for tree-structured graphs:

1. The *sum-product algorithm* for efficient marginal inference. That is, computing marginals of the form $p(x_i)$.

2. The *max-product algorithm* for MAP inference. That is, computing queries of the form $\max_{x_1,\ldots,x_n} p(x_1, \ldots, x_n)$.

Variable Elimination as Message Passing

First, consider what happens if we run the variable elimination algorithm on a tree in order to compute a marginal $p(x_i)$. We can find an optimal ordering for this problem by rooting the tree at x_i and iterating through the nodes in *post-order* — where a post-order traversal of a rooted tree is one that starts from the leaves and goes up the tree, such that each node is always visited after all of its children. The root is visited last.

This ordering is optimal because the largest clique formed during variable elimination will be of size two. At each step, we will eliminate a node x_j. This will involve computing the factor

$$\tau_k(x_k) = \sum_{x_j} \phi(x_k, x_j)\tau_j(x_j),$$

where x_k is the parent of x_j in the tree. At a later step, x_k will be eliminated, and $\tau_k(x_k)$ will be passed up the tree to the parent x_l of x_k in order to be multiplied by the factor $\phi(x_l, x_k)$, before being marginalized out. The factor $\tau_j(x_j)$ can be thought of as a *message* that x_j sends to x_k, which summarizes all of the information from the subtree rooted at x_j. We can visualize this transfer of information using arrows on a tree (e.g., Figure 4.3).

At the end of variable elimination, x_i receives messages from all of its immediate children, marginalizes them out, and we obtain the final marginal. Now suppose that after computing $p(x_i)$, we want to compute $p(x_k)$ as well. We would again run variable elimination with x_k as the root, waiting until x_k receives all messages from its children. The key insight: the messages x_k received from x_j now will be the same as those received when x_i was the root. Further intuition for why this is true lies in the fact that there is only a single path connecting any two nodes in the tree. Thus, if we store the intermediary messages of the variable elimination algorithm, we can quickly compute other marginals as well.

A Message Passing Algorithm

A key question here is, *how exactly do we produce all the messages that we need to compute all marginal distributions?*

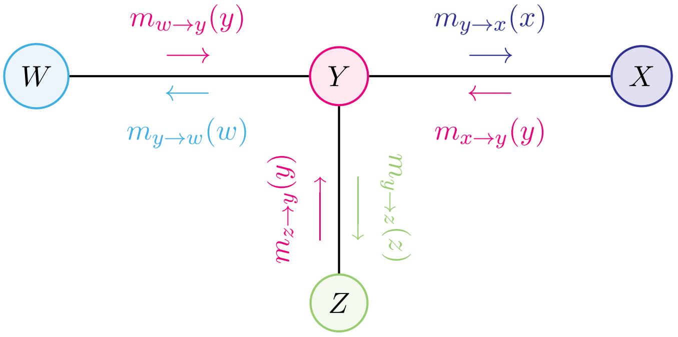


Figure 4.3: Message passing for belief propagation in an MRF. Each node pair shares messages in both direction. A message $m_{x\to y}(y)$ goes from variable X to variable Y, passing information about the state of Y. Messages are colored in correspondence with the recipient node.

The answer is very simple: a node x_i sends a message to a neighbor x_j whenever it has received messages from all nodes besides x_j. It's a fun exercise to the reader to show that in a tree, there will always be a node with a message to send, unless all the messages have been sent out. This will happen after precisely $2|\mathbf{E}|$ steps, since each edge can receive messages only twice: once from $x_i \to x_j$, and once more in the opposite direction. Finally, this algorithm will be correct because our messages are defined as the intermediate factors in the variable elimination algorithm.

4.2.1 Sum-Product Message Passing

We are now ready to formally define the belief propagation algorithm. We begin with the sum-product formulation.

The sum-product message passing algorithm is defined as follows: while there is a node x_i ready to transmit to x_j, send the message

$$m_{i\to j}(x_j) = \sum_{x_i} \phi(x_i)\phi(x_i, x_j) \prod_{x_\ell \in \mathbf{ne}(x_i)\setminus x_j} m_{\ell\to i}(x_i).$$

The notation $\mathbf{ne}(x_i) \setminus x_j$ refers to the set of nodes that are neighbors of x_i, excluding x_j. Again, observe that this message is precisely the factor τ that x_i would transmit to x_j during a round of variable elimination with the goal of computing $p(x_j)$.

Because of this observation, after we have computed all messages, we can answer any marginal query over x_i in constant time using the equation

$$p(x_i) \propto \phi(x_i) \prod_{x_\ell \in \mathbf{ne}(x_i)} m_{\ell \to i}(x_i).$$

Message Passing for Factor Trees

In addition to tree graphs, message passing can also be applied to factor graphs (Section 3.2.2), specifically in the case where the factor graph forms a tree (referred to as a *factor tree*), using a slight modification of the above procedure. Recall that a factor graph is a bipartite graph with edges going between factors and the variables in their scope. On factor graphs, we have two types of messages: variable-to-factor messages ν and factor-to-variable messages μ. Both messages require taking a product, but only the factor-to-variable messages μ require a sum:

$$\begin{aligned}
\nu_{x_i \to f_s}(x_i) &= \prod_{f_t \in \mathbf{ne}(x_i) \setminus f_s} \mu_{f_t \to x_i}(x_i) \\
\mu_{f_s \to x_i}(x_i) &= \sum_{\mathbf{ne}(f_s) \setminus x_i} f_s(\mathbf{ne}(f_s)) \prod_{x_j \in \mathbf{ne}(f_s) \setminus x_i} \nu_{x_j \to f_s}(x_j)
\end{aligned}$$

The algorithm proceeds in the same way as with undirected graphs: as long as there is a factor (or variable) ready to transmit to a variable (or factor), send the appropriate factor-to-variable (or variable-to-factor) message as defined above.

4.2.2 Max-Product Message Passing

We now introduce a second variant of the belief propagation algorithm: *max-product message passing*, which can be used to perform MAP inference queries of the form

$$p^* = \max_{x_1,\dots,x_n} p(x_1, \dots, x_n) \quad \text{or} \quad \hat{\mathbf{x}} = \operatorname*{argmax}_{x_1,\dots,x_n} p(x_1, \dots, x_n),$$

where the first expression returns the maximum value of the PDF, and the second expression instead returns the arguments that achieve this maximal value (or, equivalently, returns the MAP point estimate).

The framework that we have introduced for marginal inference allows us easily perform MAP inference as well. The key observation is that the sum and max operators both distribute over products. In general, the max operator only distributes over products of non-negative factors. By definition, however, MRF factors are non-negative. Thus, by replacing sums in marginal inference with maxes, and following the prior belief propagation algorithm, we are able to solve the MAP inference problem.

As an example, in a chain-structured MRF, we can compute the partition function (a marginal inference query) as follows:

$$\begin{aligned} Z &= \sum_{x_1} \cdots \sum_{x_n} \phi(x_1) \prod_{i=2}^{n} \phi(x_i, x_{i-1}) \\ &= \sum_{x_n} \sum_{x_{n-1}} \phi(x_n, x_{n-1}) \sum_{x_{n-2}} \phi(x_{n-1}, x_{n-2}) \cdots \sum_{x_1} \phi(x_2, x_1) \phi(x_1). \end{aligned}$$

Instead, to compute the maximum value $\tilde{p}^*$ of $\tilde{p}(x_1, \ldots, x_n)$ (a MAP inference query), we simply replace sums with maxes:

$$\begin{aligned} \tilde{p}^* &= \max_{x_1} \cdots \max_{x_n} \phi(x_1) \prod_{i=2}^{n} \phi(x_i, x_{i-1}) \\ &= \max_{x_n} \max_{x_{n-1}} \phi(x_n, x_{n-1}) \max_{x_{n-2}} \phi(x_{n-1}, x_{n-2}) \cdots \max_{x_1} \phi(x_2, x_1) \phi(x_1). \end{aligned}$$

Since both problems decompose in the same way, we can reuse all of the machinery developed for marginal inference and apply it directly to MAP inference. Note that this also applies to factor trees.

There is a small caveat: often, we want not only the maximum value of a distribution (i.e., $\max_x p(x)$), but also its most probable assignment (i.e., $\operatorname{argmax}_x p(x)$). This problem can be easily solved by keeping *back-pointers* during the optimization procedure. A back-pointer is a reference to the argument (or input value) that led to the optimal value at a given step. In the above example, we would keep a back-pointer to the best assignment to x_1 given each assignment to x_2, a pointer to the best assignment to x_2 given each assignment to x_3, and so on.

Computational Complexity

An incredible property of belief propagation on trees is that exact marginals can be computed in a number of operations that scales linearly with respect to the number of nodes in the graph. As we will see in Section 4.3, this remarkable efficiency does not hold when the input graph is no longer tree-structured.

4.3 The Junction Tree Algorithm

In our discussion on belief propagation, we assumed that the graph describing our distribution of interest is a tree. What if that is not the case? Then, inference will not be tractable. Nevertheless, we are not without options: we can transform the original graph into a tree-like form, and then run message passing on this graph.

At a high-level, the junction tree algorithm partitions the graph into clusters of variables. Internally, the variables within a cluster could be highly coupled. However, interactions *among* clusters will have a tree structure (i.e., a cluster will only be directly influenced by its neighbors in the tree). This leads to tractable global solutions if the local (cluster-level) problems can be solved exactly.

An Illustrative Example

Before we define the full junction tree algorithm, we begin with an example. Suppose that we aim to perform marginal inference on an MRF of the form

$$p(x_1, \ldots, x_n) = \frac{1}{Z} \prod_{c \in \mathbf{C}} \phi_c(\mathbf{x}_c),$$

where each factor is defined on a clique $\mathbf{x}_c$. Crucially, we assume that the cliques satisfy an assumption known as the *running intersection property*. This means there exists an ordering of the cliques $\mathbf{x}_c^{(1)}, \ldots, \mathbf{x}_c^{(n)}$ such that for any variable x_i, if $x_i \in \mathbf{x}_c^{(j)}$ and $x_i \in \mathbf{x}_c^{(k)}$, then x_i must also appear in every clique $\mathbf{x}_c^{(\ell)}$ that lies on the path between $\mathbf{x}_c^{(j)}$ and $\mathbf{x}_c^{(k)}$.

For instance, suppose that we are interested in computing the marginal probability $p(x_1)$ in a chain-structured MRF with five nodes, where each clique corresponds with a pair of nodes connected by an edge. Note that this graph satisfies the running intersection property, as each variable is contained in (at most) two adjacent cliques. We can again use a form of variable elimination to push certain variables deeper into the product of cluster potentials:

$$\phi(x_1) \sum_{x_2} \phi(x_1, x_2) \sum_{x_3} \phi(x_2, x_3) \sum_{x_4} \phi(x_3, x_4) \sum_{x_5} \phi(x_4, x_5).$$

We first sum over x_5, which creates a factor

$$\tau(x_3, x_4) = \phi(x_3, x_4) \sum_{x_5} \phi(x_4, x_5).$$

Next, x_4 gets eliminated, and so on. At each step, each cluster marginalizes out the variables that are not in the scope of its neighbor. This marginalization can also be interpreted as computing a message over the variables that the cluster shares with the neighbor.

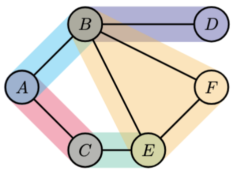
The running intersection property is what enables us to push sums in all the way to the last factor. We can eliminate x_5 because we know that only the last cluster will carry this variable: since it is not present in the neighboring cluster, it cannot be anywhere else in the graph without violating the running intersection property.

Junction Trees

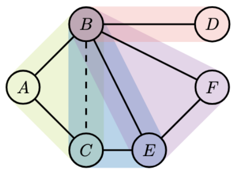
The core idea of the junction tree algorithm is to turn a graph into a tree of clusters that are amenable to the variable elimination algorithm (Figure 4.4). Then, we can simply perform message passing on this tree.

Suppose we have an undirected graphical model $\mathcal{G}$. If the model is directed, we consider the corresponding (chordalized) moral graph (Figure 4.5). A junction tree $\mathcal{T} = (\mathbf{C}, \mathbf{E}_{\mathcal{T}})$ over $\mathcal{G} = (\mathbf{X}, \mathbf{E}_{\mathcal{G}})$ is a tree whose nodes $c \in \mathbf{C}$ represent node clusters or subsets $\mathbf{x}_c \subseteq \mathbf{X}$. More specifically, these clusters are *maximal cliques* (Definition 2.36).[1] When we visualize junction trees, we often represent these clusters along with their *sepsets* (i.e., separation sets), which are the variables forming the intersection of the scopes of two neighboring cliques. A valid junction tree must satisfy the following properties:

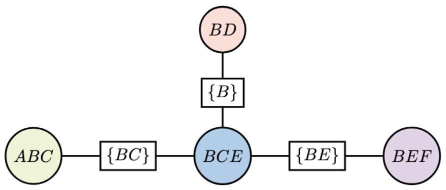
[1]We consider the moral graph when the initial graph is directed because moralization ensures that all variables which participate in a given factor belong to a single clique.



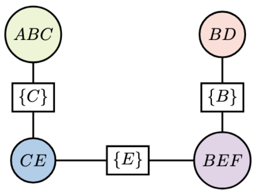
(a) INPUT GRAPH $\mathcal{G}$



(b) MAXIMAL ELIMINATION CLIQUES



(c) VALID JUNCTION TREE $\mathcal{T}$ ✓



(d) INVALID JUNCTION TREE ✗

Figure 4.4: MRF with graph $\mathcal{G}$ and corresponding junction tree $\mathcal{T}$. In MRF **(a)**, potentials are denoted by colored highlights. In chordalized graph **(b)**, highlights now correspond to the maximal elimination cliques obtained by variable elimination. Graph **(c)** demonstrates a *valid* junction tree for $\mathcal{G}$. Circular nodes in tree $\mathcal{T}$ denote maximal elimination cliques. Rectangular nodes in $\mathcal{T}$ denote *sepsets* (i.e., "separation sets"), which are sets of variables shared by neighboring clusters. Graph **(d)** represents an *invalid* junction tree for $\mathcal{G}$, as it violates the running intersection property with respect to variable B. Adapted in part from notes by Mark Paskin.

1. *Family preservation.* For each factor ϕ, there is a cluster c such that $\text{Scope}[\phi] \subseteq \mathbf{x}_c$.

2. *Running intersection.* For every pair of clusters $c^{(i)}, c^{(j)}$, every cluster on the path between $c^{(i)}, c^{(j)}$ contains $\mathbf{x}_c^{(i)} \cap \mathbf{x}_c^{(j)}$. This is sometimes referred to as the *junction tree property.*

Thus, we reach the following definition.

Definition 4.1 (Junction tree)**.** A graph $\mathcal{T} = (\mathbf{C}, \mathbf{E}_{\mathcal{T}})$ is a junction tree for graph $\mathcal{G} = (\mathbf{X}, \mathbf{E}_{\mathcal{G}})$ if and only if

1. $\mathcal{T}$ is tree-structured.
2. $\mathbf{C}$ is the set of maximal cliques of $\mathcal{G}$.
3. The running intersection property holds.

Note that we can always find a trivial junction tree with one node containing all the variables in the original graph. However, such trees are not useful for inference because they will not result in efficient marginalization algorithms.

Finding Good Junction Trees

Optimal junction trees are those that make the clusters as small and modular as possible. Unfortunately, it is NP-hard to find the optimal junction tree in the general case. A special case where we *can* efficiently identify the optimal junction tree is when $\mathcal{G}$ is itself a tree. Then, we can define a cluster for each edge in the tree. It is not hard to check that the result satisfies the above definition.

Although finding optimal junction trees is NP-hard, there are practical approaches for finding *good* junction trees for the general case.

- *By hand.* Typically, our models will have a very regular structure for which there will be an obvious solution. For example, if our model is a grid, then the clusters will be associated with pairs of adjacent rows (or columns) in the grid.
- *Using variable elimination.* The variable elimination algorithm can be used to identify the maximal cliques that will become the nodes in our junction tree.

We will elaborate on the latter approach, starting with the following propositions. Recall the notion of a *chordal graph*, where the longest minimal cycle is a triangle (Definition 2.39).

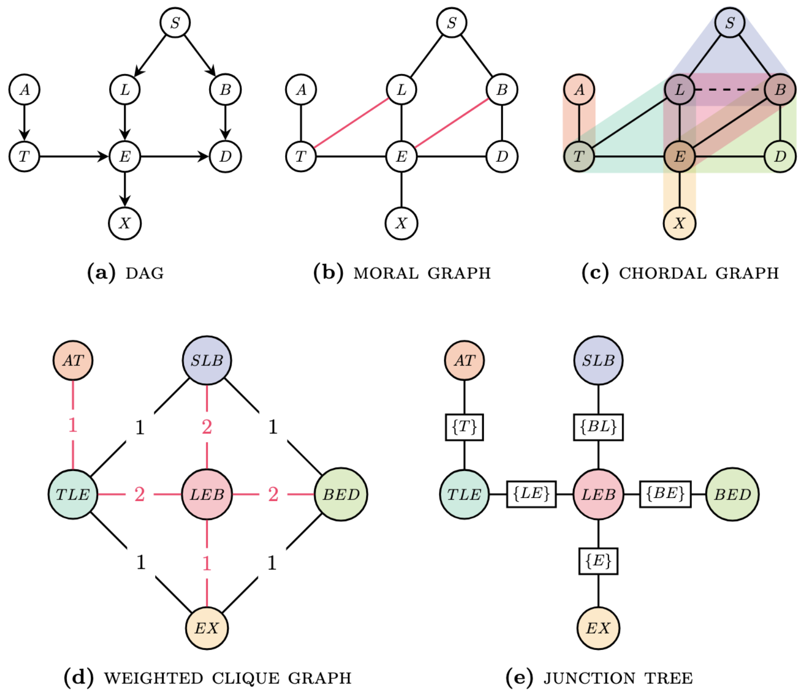


Figure 4.5: Transforming the Asia DAG (Lauritzen and Spiegelhalter, 1988) into a junction tree, per Algorithm 2. We begin with the original DAG (**a**) and obtain the undirected moral graph (**b**). We then chordalize the moral graph to identify maximal cliques (**c**; chord denoted by a dashed edge; cliques in highlights). We transform the chordal graph into a weighted clique graph, where each node represents a maximal elimination clique and edge weights are assigned according to the cardinality of the sepset (**d**). From this, we can obtain the maximum weight spanning tree (pink edges). Finally, we have our junction tree (**e**).

Proposition 4.1. Any chordal graph has a corresponding junction tree. Furthermore, any graph with a junction tree must be chordal.

Proposition 4.2. Any chordal graph with n nodes has at most n maximal cliques. Further, the chordal graph with n nodes and n maximal cliques is the graph with no edges, and a connected chordal graph has at most $n - 1$ maximal cliques.

For proof of Propositions 4.1 and 4.2, see Chapter 3 of Vandenberghe and Andersen (2015). Following from these observations, we can see that it is possible to identify the maximal cliques (and thus the junction tree) of a chordal graph in polynomial time. In fact, every maximal clique is also an *elimination clique* for a given elimination ordering in variable elimination. Thus, we can run variable elimination on a chordal graph to efficiently discover the maximal cliques.

But what if our original graph is not chordal? Fortunately, running variable elimination on a non-chordal graph will *chordalize* it along the way. After variable elimination, the reconstituted graph is guaranteed to be chordal — and, therefore, guaranteed to have a corresponding junction tree. From there, finding a junction tree equates to finding a *maximum weight spanning tree* over the complete cluster graph of the maximal elimination cliques (Figure 4.5). Maximum weight spanning trees can be efficiently identified using various algorithms (e.g., Kruskal's algorithm (Kruskal, 1956) or Prim's algorithm (Prim, 1957); see Kleinberg and Tardos 2006).

Note that multiple valid elimination orderings exist, and the maximum weight spanning tree can be non-unique. Thus, multiple valid junction trees can result, depending on the selected elimination order and the selected spanning tree.

In sum, we can obtain a valid junction tree using the procedure outlined in Algorithm 2. In Figure 4.4, we illustrate both a valid and invalid junction tree for a given MRF.

The Junction Tree Algorithm

We now define the junction tree algorithm and explain why it works. At a high-level, this algorithm implements a form of message passing on the junction tree, which will be equivalent to variable elimination (in the same way that belief propagation is equivalent to variable elimination).

More precisely, let us define the potential $\psi_c(\mathbf{x}_c)$ of each cluster c as the product of all the factors ϕ in $\mathcal{G}$ that have been assigned to c. By the family preservation property, this is well-defined, and we can assume that our distribution is of the form

$$p(x_1, \ldots, x_n) = \frac{1}{Z} \prod_{c \in \mathbf{C}} \psi_c(\mathbf{x}_c).$$

Then, at each step of the algorithm, we choose a pair of adjacent

Algorithm 2 *Obtaining a junction tree*

```
1: If the original graph is directed, obtain the moral graph (Definition 2.38).
2: Select a node ordering. Chordalize the graph (Definition 2.39) by running
   variable elimination on this ordering (Algorithm 1).
3: Construct a complete cluster graph over the maximal elimination cliques
   resulting from variable elimination.
4: Weight each edge by the cardinality of the sepset corresponding to it. This
   yields a weighted clique graph.
5: Construct a maximum weight spanning tree from the weighted clique
   graph. The resulting graph is a junction tree.
```

Algorithm 3 *Junction tree algorithm*

```
1: Input a valid junction tree $\mathcal{T}$ corresponding to original graph $\mathcal{G}$ (e.g., as
   obtained by Algorithm 2).
2: Define potentials on $\mathcal{T}$ (i.e., singleton potentials and edge potentials).
3: Run the sum-product algorithm on $\mathcal{T}$ (Algorithm 1).
4: Compute marginals for each node in $\mathcal{G}$ using the maximal clique marginals
   obtained at Line 3.
```

clusters $c^{(i)}, c^{(j)}$ in $\mathcal{T}$ and compute a message whose scope is the sepset $\mathbf{S}_{ij}$ between the two clusters:

$$m_{i\to j}(\mathbf{S}_{ij}) = \sum_{\mathbf{x}_c \setminus \mathbf{S}_{ij}} \psi_c(\mathbf{x}_c) \prod_{x_\ell \in \mathbf{ne}(x_i)\setminus x_j} m_{\ell\to i}(\mathbf{S}_{\ell i}).$$

We choose $c^{(i)}, c^{(j)}$ only if $c^{(i)}$ has received messages from all of its neighbors except $c^{(j)}$. Just as in belief propagation, this procedure will terminate in exactly $2|\mathbf{E}_{\mathcal{T}}|$ steps. After it terminates, we will define the belief of each cluster based on all the messages that it receives

$$\beta_c(\mathbf{x}_c) = \psi_c(\mathbf{x}_c) \prod_{x_\ell \in \mathbf{ne}(x_i)} m_{\ell\to i}(\mathbf{S}_{\ell i}).$$

These updates are often referred to as *Shafer-Shenoy* (Shafer and Shenoy, 1990). After all the messages have been passed, beliefs will be proportional to the marginal probabilities over their scopes, i.e., $\beta_c(\mathbf{x}_c) \propto p(\mathbf{x}_c)$. We can answer queries of the form $\tilde{p}(x)$ for $x \in \mathbf{x}_c$ by marginalizing out the variable in its belief

$$\tilde{p}(x) = \sum_{\mathbf{x}_c \setminus x} \beta_c(\mathbf{x}_c).$$

Readers familiar with combinatorial optimization will recognize this as a special case of dynamic programming on a tree decomposition of a graph with bounded treewidth. To get the actual (normalized) probability, we divide by the partition function Z which is computed by summing all the beliefs in a cluster, $Z = \sum_{\mathbf{x}_c} \beta_c(\mathbf{x}_c)$.

Note that this algorithm makes it obvious why we want small clusters: the running time will be exponential in the size of the largest cluster (if only because we can need to marginalize out variables from the cluster, which often must be done using brute force). This is why a junction tree of a single node containing all the variables is not useful: it amounts to performing full brute-force marginalization.

Once we have obtained a junction tree via Algorithm 2, we can perform the full junction tree algorithm, as in Algorithm 3.

Variable Elimination Over a Junction Tree

Why does this method work? First, let us convince ourselves that running variable elimination with a certain ordering is equivalent to performing message passing on the junction tree. Then, we will see that the junction tree algorithm is just a way of precomputing these messages and using them to answer queries.

Suppose we are performing variable elimination to compute $\tilde{p}(x')$ for some variable x', where $\tilde{p} = \prod_{c \in \mathbf{C}} \psi_c$. Let $c^{(i)}$ be a cluster containing x'. We will perform variable elimination with the ordering given by the structure of the tree rooted at $c^{(i)}$. For the MRF in Figure 4.4, say that we choose to eliminate variable B, and we set $\{A, B, C\}$ as the root cluster.

First, we pick a set of variables x_{-k} in a leaf $c^{(j)}$ of $\mathcal{T}$ that does not appear in the sepset S_{kj} between $c^{(j)}$ and its parent $c^{(k)}$ (if there is no such variable, we can multiply $\psi(\mathbf{x}_c^{(j)})$ and $\psi(\mathbf{x}_c^{(k)})$ into a new factor with a scope not larger than that of the initial factors). In our example, we can pick the variable f in the factor $\{B, E, F\}$.

Then we marginalize out x_{-k} to obtain a factor $m_{j \to k}(\mathbf{S}_{ij})$. We multiply $m_{j \to k}(\mathbf{S}_{ij})$ with $\psi(\mathbf{x}_c^{(k)})$ to obtain a new factor $\tau(\mathbf{x}_c^{(k)})$. Doing so, we have effectively eliminated the factor $\psi(\mathbf{x}_c^{(j)})$ and the unique variables it contained. In the running example, we can sum out f and the resulting factor over $\{B, E\}$ may be folded into $\{B, C, E\}$.

Note that the messages computed in this case are exactly the same as those of junction tree. In particular, when $c^{(k)}$ is ready to send its message, it will have been multiplied by $m_{\ell \to k}(\mathbf{S}_{ij})$ from all neighbors except its parent, which is exactly how junction tree sends its message.

Repeating this procedure eventually produces a single factor $\beta(\mathbf{x}_c^{(i)})$, which is our final belief. Since variable elimination implements the messages of the junction tree algorithm, $\beta(\mathbf{x}_c^{(i)})$ will correspond to the junction tree belief. Assuming we have convinced ourselves in the previous section that variable elimination works, we know that this belief will be valid.

Formally, we can prove correctness of the junction tree algorithm through an induction argument on the number of factors ψ; we will leave this as an exercise to the reader. The key property that makes this argument possible is the running intersection property: it assures us that it's safe to eliminate a variable from a leaf cluster that is not found in that cluster's sepset. By the running intersection property, this cannot occur anywhere except that one cluster.

The important thing to note is that if we now set $c^{(k)}$ to be the root of the tree (e.g., if we set $\{B, C, E\}$ to be the root), the message it will receive from $c^{(j)}$ (or from $\{B, E, F\}$ in our example) will not change. Hence, the caching approach we used for the belief propagation algorithm extends immediately to junction trees. The algorithm we formally defined above implements this caching.

Computational Complexity of the Junction Tree Algorithm

Let $\mathbf{X}$ be the nodes in our original graph $\mathcal{G}$. Both the number of cliques in the corresponding junction tree and the number of messages that we must compute in Algorithm 3 are $O(|\mathbf{X}|)$. The time complexity and space complexity of the junction tree algorithm are dominated by the treewidth (i.e., the size of the largest clique in the junction tree over all possible elimination orderings). In the general case, these will be exponential in the treewidth. Recall that the largest maximal clique size of a chordal graph will depend on the elimination ordering, such that some orderings are more favorable than others.

4.4 Exact MAP Inference

In the final section of this chapter, we shift focus from general-purpose inference algorithms to the specific problem of MAP inference in graphical models. While earlier sections introduced algorithms that are applicable to both marginal and MAP inference, there exist special cases in which MAP inference can be performed more efficiently. This section discusses the computational complexity of MAP inference and presents an example of an efficient exact algorithm based on graph cuts.

As an example, consider MAP inference in an MRF with distribution p (Equation 3.2), which corresponds to the following optimization problem:

$$\max_{\mathbf{x}} \log p(\mathbf{x}) = \max_{\mathbf{x}} \sum_c \theta_c(\mathbf{x}_c) - \log Z,$$

where $\theta_c(\mathbf{x}_c) = \log \phi_c(\mathbf{x}_c)$. We will now consider several efficient methods for solving this optimization problem. See Chapter 13 in Koller and Friedman (2009) for a thorough discussion of MAP inference.

4.4.1 The Challenges of MAP Inference

In a way, MAP inference is easier than marginal inference. One reason for this is that the intractable partition constant $\log Z$ does not depend on $\mathbf{x}$ and can be ignored. In the example above, for instance, we only need to compute

$$\operatorname*{argmax}_{\mathbf{x}} \sum_c \theta_c(\mathbf{x}_c).$$

Marginal inference can also be seen as computing and summing all assignments to the model, one of which is the MAP assignment. If we replace summation with maximization, we can also find the assignment with the highest probability. However, there exist more efficient methods than this sort of enumeration-based approach.

Note, however, that MAP inference is still not an easy problem in the general case. The above optimization objective includes many intractable problems as special cases (Shimony, 1994), such as 3-SAT. We can reduce 3-SAT to MAP inference by constructing for each clause $c = (x \vee y \vee \neg z)$ a factor $\theta_c(x, y, z)$ that equals one if x, y, z satisfy clause c, and equals zero otherwise. Then, the 3-SAT instance is satisfiable if and only if the value of the MAP assignment equals the number of clauses. We can also use a similar construction to prove that marginal inference is NP-hard. The high-level idea is to add an additional variable X that equals 1 when all the clauses are satisfied, and zero otherwise. The marginal probability will be greater than zero if and only if the 3-SAT instance is satisfiable.

Nonetheless, we will see that the MAP problem is easier than general inference, in the sense that there are some models in which MAP inference can be solved in polynomial time while general inference is NP-hard.

Illustrative Examples

Many interesting examples of MAP inference are instances of *structured prediction*, which involves doing inference in a CRF $p(\mathbf{y} \mid \mathbf{x})$:

$$\arg\max_{\mathbf{y}} \log p(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_c \theta_c(\mathbf{y}_c, \mathbf{x}_c).$$

We discussed structured prediction in detail when we covered CRFs in Section 3.2.3. Recall that our main example was handwriting recognition, in which we are given images of characters in the form of pixel matrices $\mathbf{x}_i \in [0,1]^{d\times d}$ (Figure 3.14). MAP inference in this setting amounts to jointly recognizing the most likely word $(y_i)_{i=1}^n$ encoded by the images.

Another example of MAP inference is image segmentation. Image segmentation is used frequently in computer vision applications (e.g., medical image analysis). Here, we are interested in locating an entity in an image and labeling all its pixels (e.g., the bird in Figure 4.6). Our input $\mathbf{x}_i \in [0,1]^{d\times d}$ is a matrix of image pixels, and our task is to predict the label $\mathbf{y}_i \in \{0,1\}^{d\times d}$, indicating whether each pixel encodes the object we want to recover. Intuitively, neighboring pixels should have similar values in $\mathbf{y}_i$ (e.g., pixels associated with the bird should form one continuous shape, rather than white noise).

4.4.2 Exact MAP Inference with Graph Cuts

We start our discussion with an efficient exact MAP inference algorithm called *graph cuts* for certain models over binary-valued variables ($X_i \in$

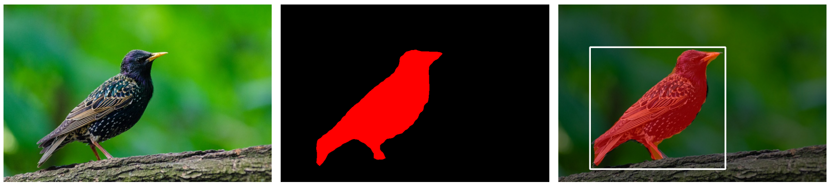


Figure 4.6: An illustration of the image segmentation problem, where a segmentation mask is used to partition the image into *bird* and *background*. Original photograph taken by Mathias Appel (public domain) with segmentation performed in Python.

$\{0, 1\}$). Unlike some methods (e.g., the junction tree algorithm), this algorithm is tractable even when the model has large treewidth.

A graph cut in an undirected graph $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ is a partition of $\mathbf{V}$ into two disjoint sets $\mathbf{V}_s$ and $\mathbf{V}_t$. When each edge $(v_1, v_2) \in \mathbf{E}$ is associated with a nonnegative cost, $cost(v_1, v_2)$, the cost of a graph cut is the sum of the costs of the edges that cross between the two partitions:

$$cost(\mathbf{V}_s, \mathbf{V}_t) = \sum_{v_1 \in \mathbf{V}_s,\ v_2 \in \mathbf{V}_t} cost(v_1, v_2).$$

The *min-cut* problem is to find the partition $\mathbf{V}_s, \mathbf{V}_t$ that minimizes the cost of the graph cut (Figure 4.7). The fastest algorithms for computing min-cuts in a graph take $O(|\mathbf{E}|\ |\mathbf{V}|\ \log|\mathbf{V}|)$ or $O(|\mathbf{V}|^3)$ time, and we refer readers to algorithms textbooks for details on their implementation (Cormen *et al.*, 2022).

Now, we show a reduction of MAP inference on a particular class of MRFs to the min-cut problem. Suppose we are given an MRF over binary variables with pairwise factors in which edge energies (i.e., negative log-edge factors) have the form

$$E_{uv}(x_u, x_v) = \begin{cases} 0 & \text{if } x_u = x_v \\ \lambda_{uv} & \text{if } x_u \neq x_v \end{cases}$$

where $\lambda_{uv} \geq 0$ is a cost that penalizes edge mismatches. Assume also that each node u has a unary potential described by an energy function $E_u(x_u)$. Thus, the full distribution is

$$p(\mathbf{x}) = \frac{1}{Z} \exp\left[-\sum_u E_u(x_u) - \sum_{u,v \in \mathbf{E}} E_{uv}(x_u, x_v)\right].$$

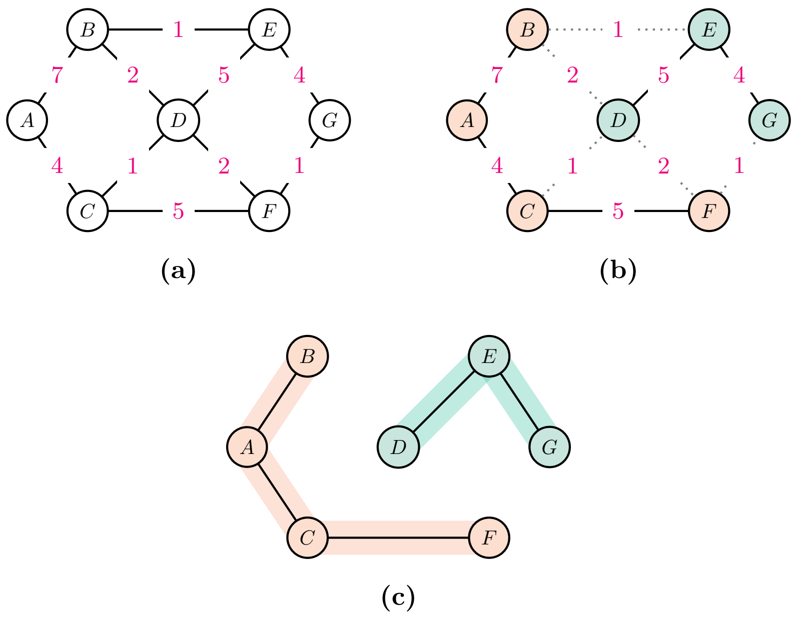


Figure 4.7: Min-cut in an MRF $\mathcal{G} = (\mathbf{V}, \mathbf{E})$. Using the edge weights in $\mathcal{G}$ (**a**), we can identify the cut with minimum cost (dotted edges in **b**). In this MRF, the resulting node partition is $\mathbf{V}_s = \{A, B, C, F\}$ and $\mathbf{V}_t = \{D, E, G\}$ (**c**).

Thus, MAP inference is equivalent to minimizing the energy:

$$\operatorname*{argmax}_{\mathbf{x}} p(\mathbf{x}) = \operatorname*{argmin}_{\mathbf{x}} \left[\sum_u E_u(x_u) + \sum_{u,v \in \mathbf{E}} E_{uv}(x_u, x_v) \right].$$

For each node u, we can normalize its energies such that $E_u \geq 0$, and either $E_u(0) = 0$ or $E_u(1) = 0$. Specifically, we replace E_u with $E'_u = E_u - \min E_u$, which is equivalent to multiplying the unnormalized probability distribution by a nonnegative constant $e^{\min E_u}$. This does not change the probability distribution. For example, we would replace

$$\begin{array}{cc} x_u & E_u(x_u) \\ \hline 0 & 4 \\ 1 & -5 \end{array} \quad \rightarrow \quad \begin{array}{cc} x_u & E'_u(x_u) \\ \hline 0 & 9 \\ 1 & 0 \end{array}.$$

The motivation for this model comes from image segmentation. We are looking for an assignment that minimizes the energy, which (among other things) tries to reduce discordance between adjacent variables.

We can formulate energy minimization in this type of model as a min-cut problem in an augmented graph $\mathcal{G}'$:

- We construct $\mathcal{G}'$ by adding special source and sink nodes s, t to our PGM graph.
- The node s is connected to nodes u with $E_u(0) = 0$ by an edge with weight $E_u(1)$.
- The node $\mathcal{T}$ is connected to nodes v with $E_v(1) = 0$ by an edge with weight $E_v(0)$.
- Finally, all the edges of the original graph get E_{uv} as their weight.

By construction, the cost of a minimal cut in this graph equals the minimum energy in the model. In particular, all nodes on the s side of the cut receive an assignment of 0, and all nodes on the $\mathcal{T}$ side receive an assignment of 1. The edges between the nodes that disagree are precisely the ones in the minimal cut.

Similar techniques can be applied in slightly more general types of models with a certain type of edge potentials that are called *submodular*. We refer the reader to Section 13.6 in Koller and Friedman (2009) for more details. In Chapter 5, we will revisit MAP inference in the context of approximation methods.

Further Reading

- V. Lepar and P. P. Shenoy. (1998). "A comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer architectures for computing marginals of probability distributions". In: *Proceedings of the Fourteenth conference on Uncertainty in Artificial Intelligence*. 328–337.
- J. S. Yedidia *et al.* (2003). "Understanding belief propagation and its generalizations". *Exploring artificial intelligence in the new millennium*. 8(236–239): 0018–9448.
- Chapter 8.4 in C. M. Bishop. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Chapter 9 in D. Koller and N. Friedman. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Chapter 20 in K. P. Murphy. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.