An Introduction to Probabilistic Graphical Models

Michael I. Jordan
University of California, Berkeley

June 30, 2003

Chapter 4

Probability Propagation and Factor Graphs

In this chapter we describe an algorithm for probabilistic inference known as the *sum-product*, or *belief propagation*, algorithm. The algorithm is closely related to the elimination algorithm, and indeed we will derive it from the perspective of elimination. The algorithm goes significantly beyond the elimination algorithm, however, in that it can compute all single-node marginals (for certain classes of graphs) rather than only a single marginal.

It is important to be clear that we are also taking a step backward in this chapter—while the elimination algorithm is applicable to arbitrary graphs, the sum-product algorithm is designed to work only in trees (or in the various "tree-like" graphs that we discuss in this chapter). Despite this step backward, there are at least three reasons why the sum-product algorithm overall represents significant progress: (1) Trees are important graphs. Indeed, a significant fraction of the classical literature on graphical models was entirely restricted to trees, and many of these classical applications require the ability to compute all singleton marginals. Examples include the hidden Markov model of Chapter 12 and the state-space model of Chapter 15. (2) The sum-product algorithm provides new insights into the inference problem, insights which will eventually allow us to provide a general solution to the exact inference problem (the *junction tree algorithm* of Chapter 17). The sum-product algorithm essentially involves an efficient "calculus of intermediate factors," which recognizes that many of the same intermediate factors are used in different elimination orderings. The junction tree algorithm extends this calculus to general graphs, by essentially combining the key ideas of the sum-product algorithm and the elimination algorithm. (3) While our focus in the current chapter is exact inference, the sum-product algorithm also provides the basis of a class of *approximate* inference algorithms for general graphs, as we discuss in Chapter 20.

Another goal of the current chapter is to introduce *factor graphs*, an alternative graphical representation of probabilities that is of particular value in the context of the sum-product algorithm. In particular, we will show that the factor graph approach provides an elegant way to handle various general "tree-like" graphs, including "polytrees," a class of directed graphical models in which nodes have multiple parents.

Finally, we also broaden our agenda in the current chapter, moving beyond the problem of com-

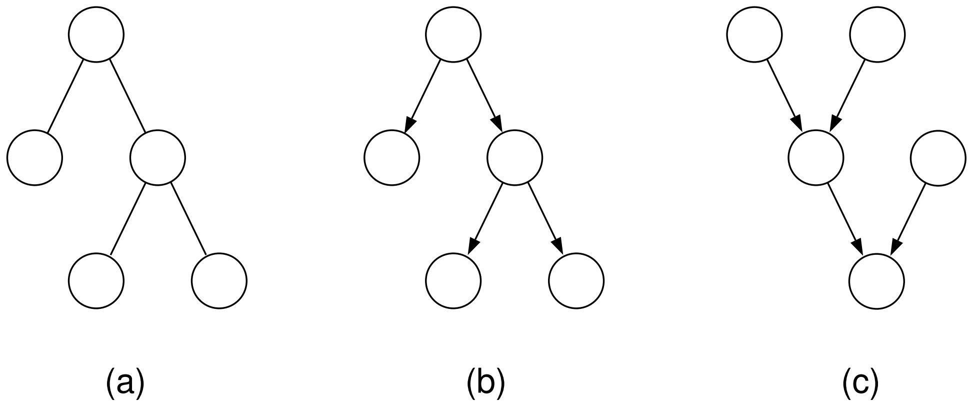


Figure 4.1: (a) An undirected tree. (b) A directed tree. (c) A polytree.

puting marginal and conditional probabilities to the problem of computing *maximum a posteriori probabilities*. We show that this problem can be solved via an algorithm that is closely related to the sum-product algorithm.

The chapter is organized as follows. In Section 4.1 we begin with a discussion of probabilistic inference on trees, treating both the directed case and the undirected case. Section **??** introduces *factor graphs*, discusses the relationships with directed and undirected graphs, and develops the sum-product algorithm for factor graphs. We discuss polytrees in Section 4.2.4, and discuss algorithms for computing maximum a posteriori probabilities in Section 4.3.

4.1 Probabilistic inference on trees

In this section we describe an inference algorithm for trees. Let us first clarify exactly what is meant by a "tree." In the undirected case, a tree is an undirected graph in which there is one and only one path between any pair of nodes. An example of an undirected tree is shown in Figure 4.1(a).[1] In the directed case, we define a tree to be any graph whose moralized graph is an undirected tree. Figure 4.1(b) shows a directed tree. Note that directed trees have a single node that has no parent—the *root node*—and that all other nodes have exactly one parent. Finally, note that the graph in Figure 4.1(c) is not a directed tree; it has nodes with multiple parents, and the resulting moralized graph has loops.

Any undirected tree can be converted into a directed tree by choosing a root node and orienting all edges to point away from the root.

From the point of view of graphical model representation and inference there is little significant difference between directed trees and undirected trees. A directed tree and the corresponding undirected tree (the tree obtained by dropping the directionality of the edges) make exactly the same set of conditional independence assertions. Morever, as we show below, the parameterizations

[1]Note that throughout the chapter we assume implicitly that our graphs are connected, and thus we have a single tree rather than a forest. This is done without loss of generality—in the case of a forest we have a collection of probabilistically independent trees, and it suffices to run an inference algorithm separately on each tree.

are essentially the same, with the undirected parameterization being slightly more flexible by not requiring potentials to be normalized (but, see Exercise **??**, any undirected representation can be readily converted to a directed one).

4.1.1 Parameterization and conditioning

Let us first consider the parameterization of probability distributions on undirected trees. The cliques are single nodes and pairs of nodes, and thus the joint probability can be parameterized via potential functions $\{\psi(x_i)\}$ and $\{\psi(x_i, x_j)\}$. In particular, we have:

$$p(x) = \frac{1}{Z}\left(\prod_{i \in \mathcal{V}} \psi(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j)\right), \tag{4.1}$$

for a tree $\mathcal{T}(\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$.

For directed trees, the joint probability is formed by taking a product over a marginal probability, $p(x_r)$, at the root node r, and conditional probabilities, $\{p(x_j \,|\, x_i)\}$, at all other nodes:

$$p(x) = p(x_r) \prod_{(i,j) \in \mathcal{E}} p(x_j \,|\, x_i), \tag{4.2}$$

where (i, j) is a directed edge such that i is the (unique) parent of j (i.e., $\{i\} = \pi_j$). We can treat such a parameterization as a special case of Eq. (4.1), and indeed it will be convenient to do so throughout this chapter. We define:

$$\begin{aligned} \psi(x_r) &= p(x_r) && (4.3)\\ \psi(x_i, x_j) &= p(x_j \,|\, x_i), && (4.4) \end{aligned}$$

for i the parent of j, and define all other singleton potentials, $\psi(x_i)$, for $i \neq r$, to be equal to one. We thereby express the joint probability for a directed tree in the undirected form in Eq. (4.1), with $Z = 1$.

Recall that we use "evidence potentials" to capture conditioning. Thus, if we are interested in the conditional probability $p(x_F \,|\, \bar{x}_E)$, for some subset E, we define evidence potentials $\delta(x_i, \bar{x}_i)$, for $i \in E$, and multiply the joint probability by the product of these potentials. This simply reduces to multiplying $\psi(x_i)$ by $\delta(x_i, \bar{x}_i)$, for $i \in E$. In particular, we define:

$$\psi_i^E(x_i) \triangleq \begin{cases} \psi_i(x_i)\delta(x_i, \bar{x}_i) & i \in E \\ \psi_i(x_i) & i \notin E, \end{cases} \tag{4.5}$$

and substitute in Eq. (4.1) to obtain:

$$p(x \,|\, \bar{x}_E) = \frac{1}{Z^E}\left(\prod_{i \in V} \psi^E(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j)\right), \tag{4.6}$$

where $Z^E = \sum_x \left(\prod_{i \in V} \psi^E(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j) \right)$. Note that the original Z vanishes.

In summary, the parameterization of unconditional distributions and conditional distributions on trees is formally identical, involving a product of potential functions associated with each node and each edge in the graph. We can thus proceed without making any special distinction between the unconditional case and the conditional case.

Are there any special features of directed trees that we lose in working exclusively with the undirected formalism? One feature of the parameterization for directed trees is that any summation of the form $\sum_{x_j} p(x_j \,|\, x_i)$ is necessarily equal to one, and does not need to be performed explicitly. Indeed, in the unconditional case, we can arrange things such that all sums are of this form, by choosing an elimination ordering that begins at the leaves and proceeds backward to the root. (This shows that the normalization factor Z is necessarily equal to one in the unconditional case). When we condition, however, the resulting product of potentials is unnormalized (the normalization factor Z^E is no longer one), and we are brought closer to the general undirected case. It is still the case that we can "prune" any subtree that contains only variables that are not conditioned on, by again eliminating backwards. We view this as an implementation detail, however, assuming that any implementation of an inference algorithm will be smart enough to prune such subtrees at the outset. We then find ourselves in a situation in which the leaves of the tree are evidence nodes, and all of the sums have to be performed explicitly. In this case, there is no essential difference between the directed case and the undirected case, and in developing the general algorithm for inference on trees, it is convenient to focus exclusively on the latter.

4.1.2 From elimination to message-passing

In this section and the following section, we derive the SUM-PRODUCT algorithm, a general algorithm for probabilistic inference on trees. The algorithm involves a simple mathematical update equation—a sum over a product of potentials—applied once for each outgoing edge at each node. We derive this update equation from the point of view of the ELIMINATE algorithm. We subsequently prove that a more general algorithm based on this update equation finds all (singleton) marginals simultaneously.

Let us begin by returning to ELIMINATE, but specializing to the case of a tree. Recall the basic structure of ELIMINATE: (1) Choose an elimination ordering I in which the query node f is the final node; (2) Place all potentials on an active list; (3) Eliminate a node i by removing all potentials referencing the node from the active list, taking the product, summing over x_i, and placing the resulting intermediate factor back on the active list. What are the special features of this procedure when the graph is a tree?

To take advantage of the recursive structure of a tree, we need to specify an elimination ordering I that respects this structure. In particular, we consider elimination orderings that arise from a *depth-first traversal* of the tree. Treat f as a root and view the tree as a directed tree by directing all edges of the tree to point away from f. We now consider any elimination ordering in which a node is eliminated only after all of its children in the directed version of the tree are eliminated. It can be easily verified that such an elimination ordering proceeds inward from the leaves, and generates elimination cliques of size at most two (showing that the tree-width of a tree is equal to

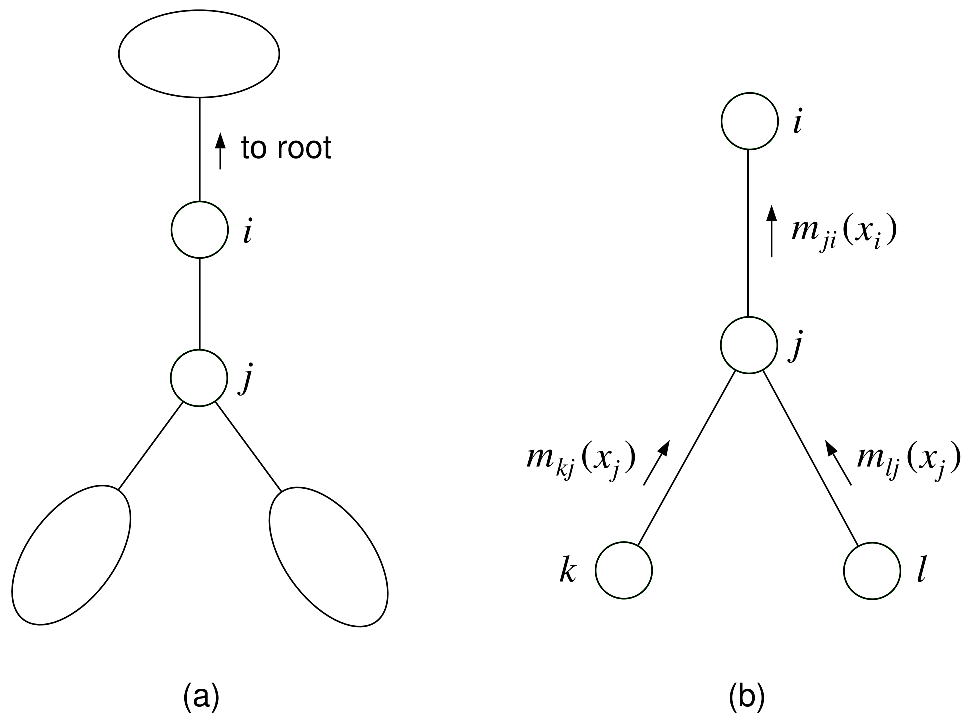


Figure 4.2: (a) A fragment of an undirected graph. Nodes i and j are neighbors, with i nearer to the root than j. (b) The messages that are created when nodes k, l and j are eliminated.

one).

Let us now consider the elimination step. Consider nodes i and j that are neighbors in the tree, where i is closer to the root than j (see Figure 4.2(a)). We are interested in the intermediate factor that is created when j is eliminated. This intermediate factor is a sum over a product of certain potentials. Which potentials are these? Clearly $\psi(x_i, x_j)$ is one of these potentials, given that it references x_j and given that i has yet to be eliminated. Also, $\psi^E(x_j)$ will appear. We can also exclude a number of possibilities. In particular, none of the potentials in the product can reference any variable in the subtree below j, given that all of these variables have already been eliminated. Moreover, none of these potentials can reference any other variable outside the subtree, due to the assumption that the graph is a tree. That is, for a node k in the subtree and a node l outside of the subtree, there can be no potential $\psi(x_k, x_l)$ in the probability model. Thus, when eliminating nodes in the subtree, we can never introduce any variable outside of the subtree into a summand and thus into an intermediate factor.

We have shown that the intermediate factor created by the sum over x_j is a function solely of x_i. Let us introduce the notation "$m_{ji}(x_i)$" to denote this term, where the first subscript denotes the variable being eliminated and the second subscript denotes the (sole) remaining neighbor of the variable (the "bucket" in the language of Section **??**). Note that the latter index is superfluous in the context of ELIMINATE—it is determined by the graph structure and the elimination ordering—but it will be needed in the context of the more general SUM-PRODUCT algorithm.

We refer to the intermediate factor $m_{ji}(x_i)$ as a "message" that j sends to i. As suggested by Figure 4.2(b), we can think of this message as "flowing" along the edge linking j to i.

Let us now consider the mathematical operation that creates the message $m_{ji}(x_i)$ in more detail. In particular, consider the potentials that are selected from the active list when we eliminate node j—the potentials that reference x_j. As we mentioned earlier, the potentials $\psi(x_i, x_j)$ and $\psi^E(x_j)$ are among the potentials selected. The other potentials that are selected are those created in earlier elimination steps in which the neighbors of node j (other than i) are eliminated. As shown in Figure 4.2(b), these steps can be viewed as creating messages $m_{kj}(x_j)$, messages that flow from each neighbor k—where $k \in \mathcal{N}(j) \backslash i$—to j.

Thus, following the protocol of the ELIMINATE algorithm, to eliminate x_j we take the product over all potentials that reference x_j and sum over x_j:

$$m_{ji}(x_i) = \sum_{x_j} \left(\psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \backslash i} m_{kj}(x_j) \right). \tag{4.7}$$

This is the intermediate factor ("message") that j sends to i.

Finally, let us consider the final node f in the elimination ordering I. All other nodes have been eliminated when we arrive at f, and thus messages $m_{ef}(x_f)$ have been computed for each of the neighbors $e \in \mathcal{N}(f)$. These messages, and the potential $\psi^E(x_f)$, are the only terms on the active list at this point. Thus, again following the protocol of ELIMINATE, we write the marginal of x_f as the following product:

$$p(x_f \,|\, \bar{x}_E) \propto \psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}(x_f), \tag{4.8}$$

where the proportionality constant is obtained by summing the right-hand side with respect to x_f.

Eqs. (4.7) and (4.8) provide a concise mathematical summary of the ELIMINATE algorithm, for the special case of a tree. Leaving behind the algorithmic details of ELIMINATE, we see that probabilistic inference essentially involves solving a coupled system of equations in the variables $m_{ji}(x_i)$. To compute $p(x_f)$, we solve these equations in an order that corresponds to a depth-first traversal of a directed tree in which f is the root.

4.1.3 The SUM-PRODUCT algorithm

In this section we show that Eqs. (4.7) and (4.8) suffice for obtaining not only a single marginal, but also for obtaining *all* of the marginals in the tree. The (somewhat magical) fact is that we can obtain all marginals by simply doubling the amount of work required to compute a single marginal. In particular, as we will show, after having passed messages inward from the leaves of the tree to an (arbitrary) root, we simply pass messages from the root back out to the leaves, again using Eq. (4.7) at each step. The net effect is that a single message will flow in both directions along each edge. Once all such messages have been computed, we invoke Eq. (4.8) independently at each node; this yields the desired marginals.

One way to understand why this algorithm works is to consider the naive approach of computing all marginals by using a different elimination ordering for each marginal. Consider in particular the tree fragment shown in Figure 4.3(a). To compute the marginal of X_1 using elimination, we eliminate X_4 and X_3, which, as we have seen, involves computing messages $m_{42}(x_2)$ and $m_{32}(x_2)$ that are sent to X_2. We subsequently eliminate X_2, which creates a message $m_{21}(x_1)$ that is sent from X_2 to X_1.

Now suppose that we wish to compute the marginal at X_2 using elimination. As shown in Figure 4.3(b), we eliminate X_4, X_3, and X_1, passing messages $m_{42}(x_2)$, $m_{32}(x_2)$ and $m_{12}(x_2)$ to X_2. The message $m_{12}(x_2)$ is new, but (crucially) $m_{42}(x_2)$ and $m_{32}(x_2)$ are the same messages as computed earlier. Similarly, if we wish to compute the marginal at X_4, as shown in Figure 4.3(c), we need a new message $m_{24}(x_4)$, but we can reuse the messages $m_{32}(x_2)$ and $m_{12}(x_2)$. In general, if we compute a message for each direction along each edge in the tree, as shown in Figure 4.3(d), we can obtain all singleton marginals.

The idea that messages can be "reused" is important. In effect we can achieve the effect of computing over all possible elimination orderings (a huge number) by computing all possible messages (a small number). This is the key insight behind the SUM-PRODUCT algorithm.

The SUM-PRODUCT algorithm is based on Eqs. (4.7), (4.8), and a "protocol" that determines when any one of these equations can be invoked. The protocol is given as follows:

Message-Passing Protocol. *A node can send a message to a neighboring node when (and only when) it has received messages from all of its other neighbors.*

There are two principal ways to implement algorithms that respect this protocol. The first (and most direct) way is to interpret the protocol as the specification of a parallel algorithm. In particular, let us view each node as a processor, and assume that the node can repeatedly poll its incoming edges for the presence of messages. For a node of degree d, whenever messages have arrived on any subset of $d-1$ edges, the node computes a message for the remaining edge and delivers the message along that edge.

An example is shown in Figure 4.4. We assume a synchronous parallel algorithm, and at each step show the messages that are delivered along the edges. Note that messages start to flow in from the leaves. Note also that when the algorithm terminates, it is the case that a pair of messages have been computed for each edge, one for each direction. Finally, note that all incoming messages are eventually computed for each node, and that Eq. (4.8) can therefore be invoked at each node to compute the node marginal.

For this algorithm to be meaningful in general, we need to insure that all messages will eventually be computed and delivered; that is, that the algorithm will never "block." We provide a proof that the protocol is non-blocking in Corollary **??** below.

We can also consider sequential implementations of the SUM-PRODUCT algorithm, in which messages are computed according to a particular "schedule." One such schedule (a schedule that is widely used in practice) is a two-phase schedule based on depth-first traversal from an arbitrary root node.[2] In the first phase, messages flow inward from the leaves toward the root (as in Section 4.1.2).

[2]The original graph may have been a directed tree, with a corresponding root node. The "root" that is designated for the purposes of the SUM-PRODUCT algorithm is unrelated to this root node.

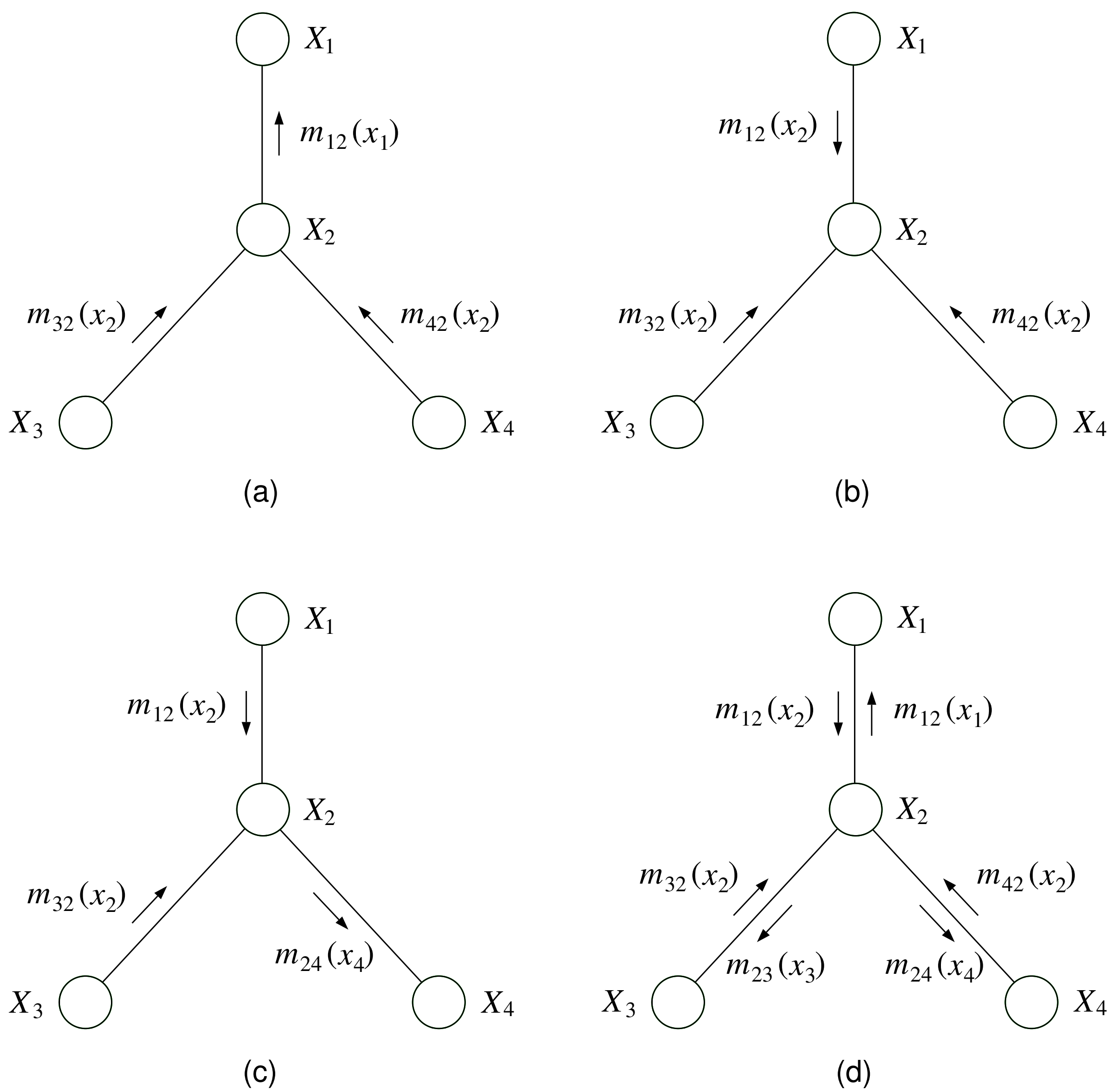


Figure 4.3: (a) The messages formed when computing the marginal of X_1. (b) The messages formed when computing the marginal of X_2. (c) The messages formed when computing the marginal of X_4. (d) All of the messages needed to compute all singleton marginals.

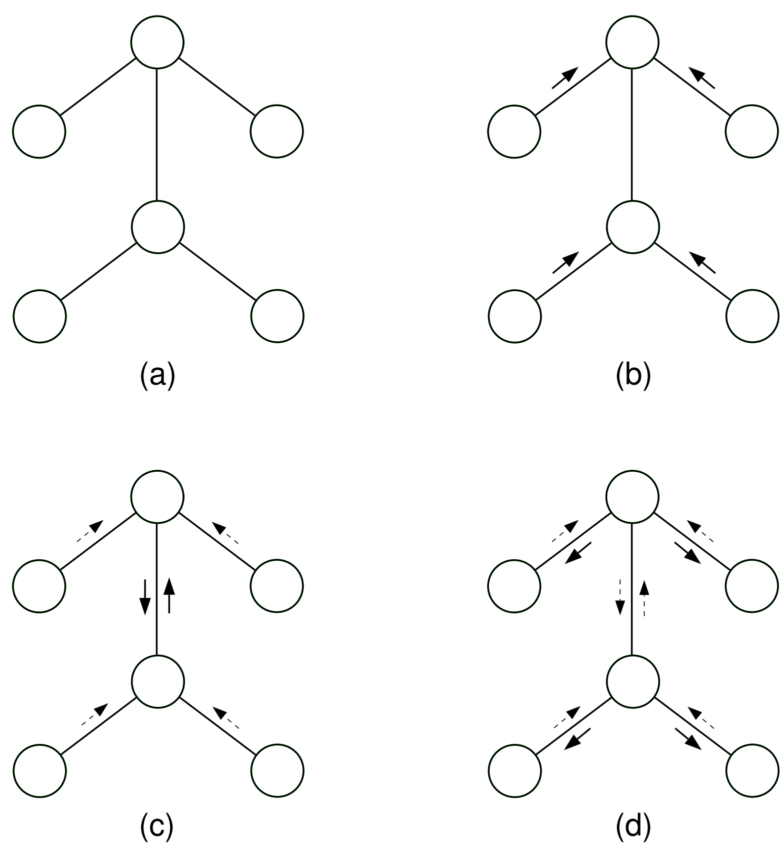


Figure 4.4: Message-passing under a synchronous parallel algorithm. The solid arrows are the messages passed at a given time step, and the dashed arrows are those passed on earlier time steps.

Sum-Product($\mathcal{T}$, E)
  Evidence(E)
  $f =$ ChooseRoot($\mathcal{V}$)
  for $e \in \mathcal{N}(f)$
    Collect(f, e)
  for $e \in \mathcal{N}(f)$
    Distribute(f, e)
  for $i \in \mathcal{V}$
    ComputeMarginal(i)

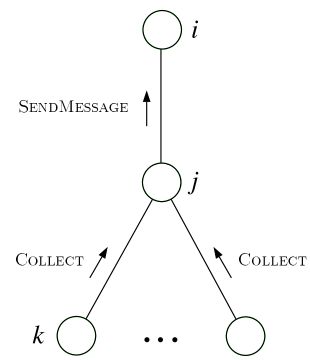
Evidence(E)
  for $i \in E$
    $\psi^E(x_i) = \psi(x_i)\delta(x_i, \bar{x}_i)$
  for $i \notin E$
    $\psi^E(x_i) = \psi(x_i)$

Collect(i, j)
  for $k \in \mathcal{N}(j)\backslash i$
    Collect(j, k)
  SendMessage(j, i)

Distribute(i, j)
  SendMessage(i, j)
  for $k \in \mathcal{N}(j)\backslash i$
    Distribute(j, k)

SendMessage(j, i)
  $m_{ji}(x_i) = \sum_{x_j} (\psi^E(x_j)\psi(x_i, x_j) \prod_{k \in \mathcal{N}(j)\backslash i} m_{kj}(x_j))$

ComputeMarginal(i)
  $p(x_i) \propto \psi^E(x_i) \prod_{j \in \mathcal{N}(i)} m_{ji}(x_i)$



1

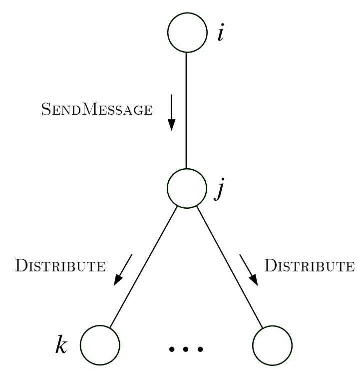


Figure 4.5: A sequential implementation of the Sum-Product algorithm for a tree $\mathcal{T}(\mathcal{V}, \mathcal{E})$. The algorithm works for any choice of root node, and thus we have left ChooseRoot unspecified. A call to Collect causes messages to flow inward from the leaves to the root. A subsequent call to Distribute causes messages to flow outward from the root to the leaves. After these calls have returned, the singleton marginals can be computed locally at each node.

In the second phase—which is initiated once all incoming messages have been received by the root node—messages flow outward from the root toward the leaves. In Figure 4.5, we show how such a schedule can be implemented via a pair of recursive function calls. In Exercise **??**, we ask the reader to show that this schedule respects the Message-Passing Protocol, and to show that the overall effect of the schedule is that a single message flows in each direction along each and every edge.

4.1.4 Proof of correctness of the SUM-PRODUCT algorithm[3]

[Section not yet written].

4.2 Factor graphs and the SUM-PRODUCT algorithm

The graphical model representations that we have discussed thus far—directed and undirected graphical models—aim at characterizing probability distributions in terms of conditional independence statements. *Factor graphs*, an alternative graphical representation of probability distributions, aim at capturing factorizations. As we have discussed (see Section **??**), while closely related, conditional independence and factorization are not exactly the same concepts. Recall in particular our discussion of the parameterization of the complete graph on three nodes. This graph makes no conditional independence assertions, and the corresponding parameterization is simply the arbitrary potential $\psi(x_1, x_2, x_3)$. However, we may be interested in endowing the potentials with algebraic structure, for example:

$$\psi(x_1, x_2, x_3) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_1, x_3), \tag{4.9}$$

for given functions f_a, f_b and f_c. Such a factorized potential defines a proper subset of the family of probability distributions associated with the complete graph, a subset which has no interpretation in terms of conditional independence. Factor graphs provide a convenient way to represent subsets of this kind.

In the following section, we introduce the general factor graph representation, and discuss its relationships to directed and undirected graphs. We then focus on the special case of *factor trees* (factor graphs that are trees), and describe the variant of the SUM-PRODUCT algorithm that is geared to factor trees.

4.2.1 Factor graphs

Given a set of variables $\{x_1, x_2, \ldots, x_n\}$, we let $\mathcal{C}$ denote a set of subsets of $\{1, 2, \ldots, n\}$. Thus, for example, given variables $\{x_1, x_2, x_3, x_4, x_5\}$, we might have $\mathcal{C} = \{\{1,3\}, \{3,4\}, \{2,4,5\}, \{1,3\}\}$. Note that $\mathcal{C}$ is a *multiset*—we allow the same subset of indices to appear multiple times. To avoid ambiguity, we therefore index the members of $\mathcal{C}$ using an *index set* $\mathcal{F}$; thus, $\mathcal{C} = \{C_s : s \in \mathcal{F}\}$.

[3]This section can be skipped without loss of continuity.

To each index $s \in \mathcal{F}$, we associate a *factor* $f_s(x_{C_s})$, a function on the subset of variables indexed by C_s. In our example, letting $\mathcal{F} = \{a, b, c, d\}$ denote the indices, the factors are $f_a(x_1, x_3)$, $f_b(x_3, x_4)$, $f_c(x_2, x_4, x_5)$ and $f_d(x_1, x_3)$.

Note also that there is no assumption that the subsets $\mathcal{C}$ correspond to cliques of an underlying graph. Indeed, at this point we do not have any graph structure in mind—$\mathcal{C}$ is just an arbitrary collection of subsets of indices.

Given a collection of subsets and the associated factors, we define a multivariate function on the variables $\{x_1, x_2, \ldots, x_n\}$ by taking the product:

$$f(x_1, x_2, \ldots, x_n) \triangleq \prod_{s=1}^{S} f_s(x_{C_s}). \tag{4.10}$$

Our goal will be to define a graphical representation of this function that will permit the efficient evaluation of *marginal functions*—functions of a single variable obtained by summing over all other variables.

Factorized functions in the form of Eq. (4.11) occur in many areas of mathematics, and the methods that we describe in this section has numerous applications outside of probability theory. Our interest, however, will be focused on factorized representations of probability distributions, and indeed the factorized probability distributions associated with directed and undirected graphical models provide examples of the general product-of-factors in Eq. (4.11).[4]

We now introduce a graphical representation of Eq. (4.11). This graphical representation—the *factor graph*—differs from directed and undirected graphical models in that it includes explicit nodes for the factors as well as the variables. We use round nodes to represent the variables and square nodes to represent the factors.

Formally, a factor graph is a bipartite graph $\mathcal{G}(\mathcal{V}, \mathcal{F}, \mathcal{E})$, where the vertices $\mathcal{V}$ index the variables and the vertices $\mathcal{F}$ index the factors. The edges $\mathcal{E}$ are obtained as follows: each factor node $s \in \mathcal{F}$ is linked to all variable nodes in the subset C_s. These are the only edges in the graph.

An example of a factor graph is shown in Figure 4.6. This graph represents the factorized function:

$$f(x_1, x_2, x_3, x_4, x_5) = f_a(x_1, x_3) f_b(x_3, x_4) f_c(x_2, x_4, x_5) f_d(x_1, x_3). \tag{4.11}$$

Note that $f_a(x_1, x_3)$ and $f_d(x_1, x_3)$ refer to the same set of variables. In an undirected graphical model these factors would be collapsed into a single potential function, $\psi(x_1, x_3)$. In a factor graph these functions are allowed to maintain a separate identity.

It will prove useful to define neighborhood functions on the nodes of a factor graph. In particular, let $\mathcal{N}(s) \subset \mathcal{V}$ denote the set of neighbors of a factor node $s \in \mathcal{F}$, and let $\mathcal{N}(i) \subset \mathcal{F}$ denote the set of neighbors of a variable node $i \in \mathcal{V}$. Note that $\mathcal{N}(s)$ refers to the indices of all variables referenced by the factor f_s, and is identical to the subset C_s introduced earlier. On the other hand, the neighborhood set $\mathcal{N}(i)$, for a variable node i, is the set of all factors that reference the variable x_i.

Directed and undirected graphical models can be readily converted to factor graphs. For example, the directed graphical model shown in Figure 4.7(a) can be represented as a factor graph

[4] The normalization factor Z in the parameterization of undirected graphical models can be treated as a factor

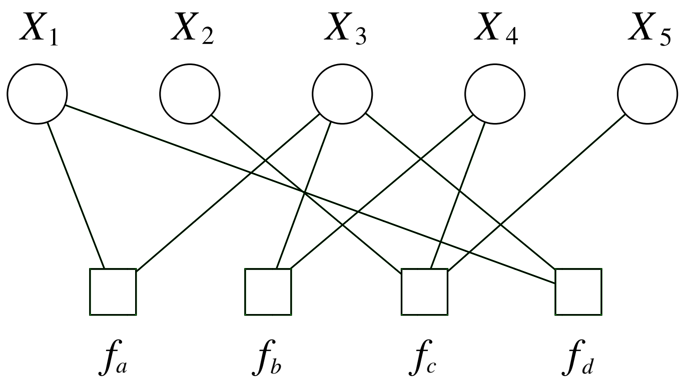


Figure 4.6: An example of a factor graph.

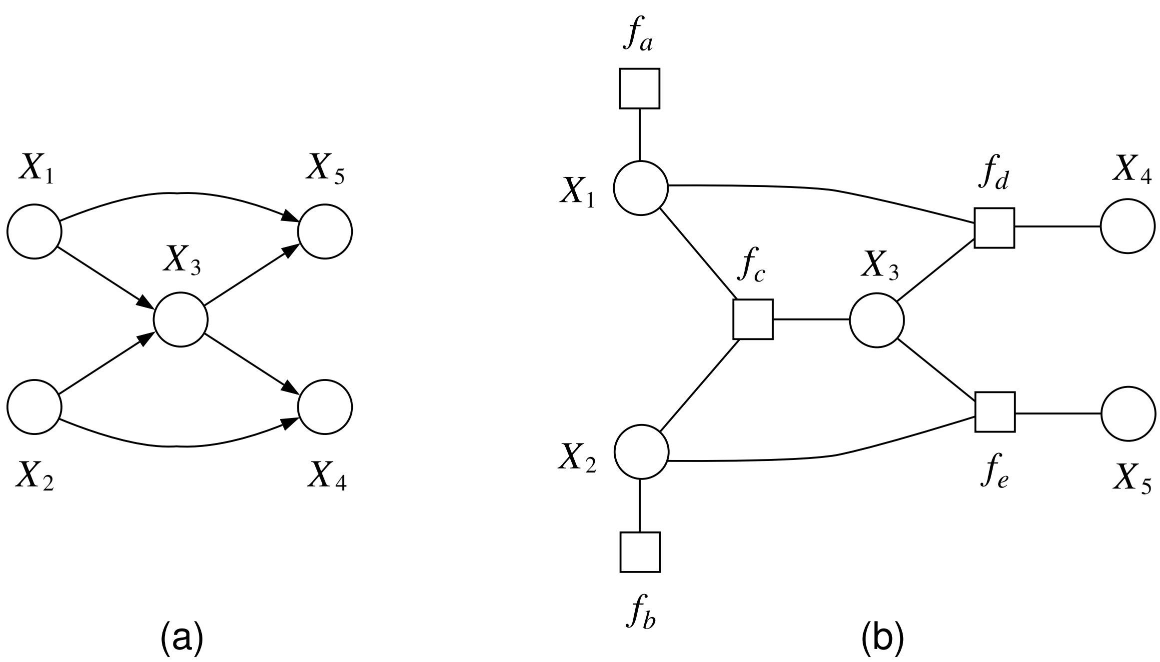


Figure 4.7: (a) A directed graphical model. (b) The corresponding factor graph. Note that there are six factor nodes, one for each local conditional probability in the directed graph.

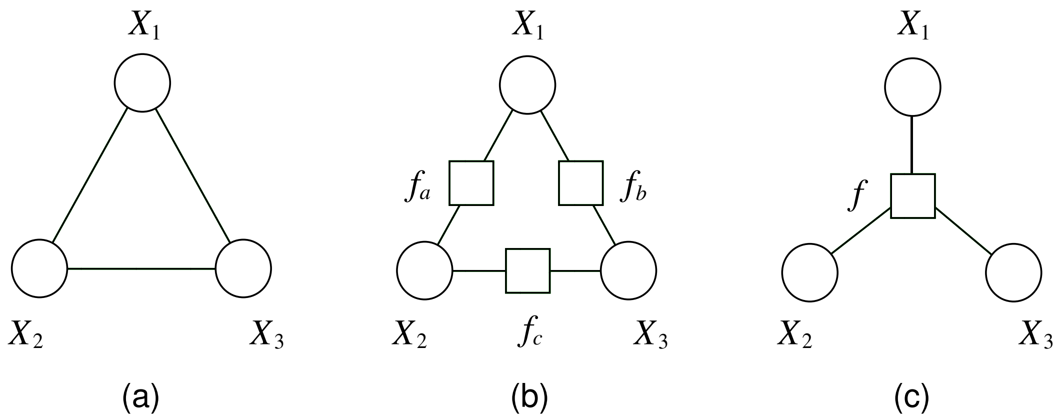


Figure 4.8: (a) An undirected graphical model provides no information about possible factorizations of the potential function associated with a given clique. (b) The factor graph corresponding to the factorized potential $\psi(x_1, x_2, x_3) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_1, x_3)$. (c) The factor graph corresponding to the non-factorized potential $\psi(x_1, x_2, x_3) = f(x_1, x_2, x_3)$.

as shown in Figure 4.7(b).[5]

By representing each factor as a node in the graph, factor graphs provide a more fine-grained representation of probability distributions than is provided by directed and undirected graphical models. In particular, returning to the complete graph on three nodes shown in Figure 4.8(a), factor graphs make it possible to display fine-grained assumptions about the parameterization: Figure 4.8(b) shows the factor graph corresponding to the general potential $\psi(x_1, x_2, x_3)$, while Figure 4.8(c) shows the factor graph corresponding to the factorized potential in Eq. (4.9).

It is worth noting that it is always possible to mimic the fine-grained representation of factor graphs within the directed and undirected formalisms, so that formally factor graphs provide no additional representational power. For example, in Figure 4.9(a) we show an undirected graph that can represent the factorization in Eq. (4.9). In this graph, we have introduced three new random variables, Z_1, Z_2, and Z_3. These variables are indicator variables picking out particular combinations of the underlying variables X_1, X_2 and X_3. Thus, for example, for binary X_1 and X_2, Z_1 would take on four possible values, one for each pair of values of X_1 and X_2, and the potential function $\psi(z_1)$ would be set equal to the corresponding value of $f_a(x_1, x_2)$. (We ask the reader to fill in the details of this construction in Exercise **??**).

Similarly, in Figure 4.9(b), we show a directed graph that mimics the factorization in Eq. (4.9). In this graph, the three new variables, W_1, W_2, and W_3, are binary variables that are always set equal to one. We set $p(W_1 = 1 \,|\, x_1, x_2)$ to the corresponding value of $f_a(x_1, x_2)$. (We again ask the reader to supply the details in Exercise **??**).

---

associated with the empty set—which is appropriate given that it is a constant.

[5]In general, in the directed case each factor is a local conditional probability, and the subsets C_s correspond to "families" consisting of a node and its parents. Given that we do not assume that the subsets C_s correspond to cliques of an underlying graph, we do not need to "moralize" in the factor graph formalism. This is consistent with the fact that the factor graph does not attempt to represent conditional independencies.

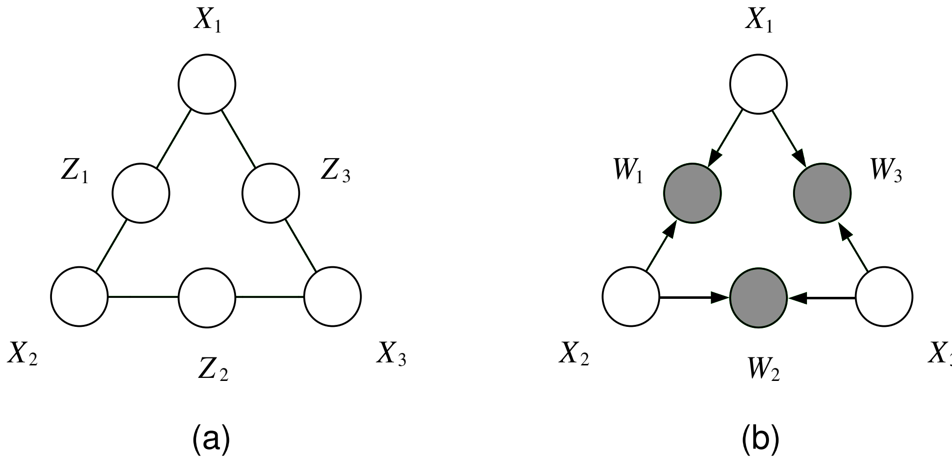


Figure 4.9: (a) An undirected graph that mimics the factorization shown in Figure 4.8(b) for appropriate choices of the indicator variables Z_i. (b) A directed graph that mimics the factorization shown in Figure 4.8(b) for appropriate choices of the indicator variables W_i.

In general, by introducing additional variables in a directed or undirected graph, we can mimic the factorization that is made explicit in the factor graph. However, this procedure is arguably rather artificial, and the factor graph representation provides a natural complement to undirected or directed graphs for situations in which a fine-grained representation of potentials is desired.

4.2.2 The Sum-Product algorithm for factor trees

We now turn to the inference problem for factor graphs. As before, our goal is to compute all singleton marginal probabilities under the factorized representation of the joint probability. In this section we show how to do this for factor graphs that are trees.

A factor graph is defined to be a *factor tree* if the undirected graph obtained by ignoring the distinction between variable nodes and factor nodes is an undirected tree. Restricting ourselves to trees, we define a variant of the Sum-Product algorithm that provides all singleton marginal probabilities for factor trees.

As in the earlier Sum-Product algorithm, we define messages that flow along the edges of the graph. In the case of factor trees, there are *two* kinds of messages: messages ν that flow from variable nodes to factor nodes, and messages μ that flow from factor nodes to variable nodes.

These messages take the following form. We first consider the messages that flow from variable nodes to factor nodes. As depicted in Figure 4.10(a), the message $\nu_{is}(x_i)$ that flows between the variable node i and the factor node s is computed as follows:

$$\nu_{is}(x_i) = \prod_{t \in \mathcal{N}(i) \backslash s} \mu_{ti}(x_i), \tag{4.12}$$

where the product is taken over all incoming messages to variable node i, other than the message from the factor node s that is the recipient of the message.

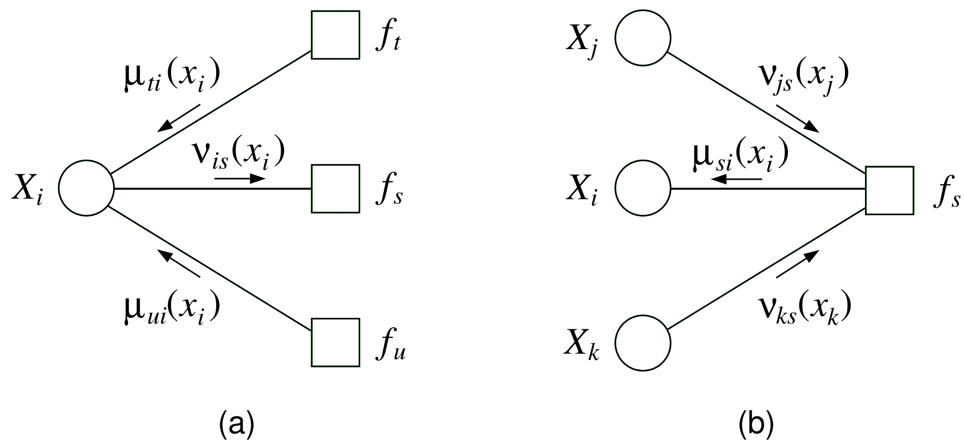


Figure 4.10: (a) The computation of the message $\nu_{is}(x_i)$ that flows from factor node s to variable node i. (b) The computation of the message $\mu_{si}(x_i)$ that flows from variable node i to factor node s.

Similarly, as shown in Figure 4.10(b), a message $\mu_{si}(x_i)$ flows between the factor node s and the variable node i. This message is computed as follows:

$$\mu_{si}(x_i) = \sum_{x_{\mathcal{N}(s)\setminus i}} \left(f_s(x_{\mathcal{N}(s)}) \prod_{j \in \mathcal{N}(s)\setminus i} \nu_{js}(x_j) \right). \tag{4.13}$$

Note that the product is taken over all incoming messages to factor node s, other than the message from the variable node i that is the recipient of the message.

Thus we have a coupled set of equations for a set of messages. As in our earlier SUM-PRODUCT algorithm, a full specification of the algorithm requires a determination of when a given equation can be invoked. The protocol turns out to be exactly the same as the earlier protocol:

Message-Passing Protocol. *A node can send a message to a neighboring node when (and only when) it has received messages from all of its other neighbors.*

In the factor tree case, the protocol applies to both variable nodes and factor nodes.

Finally, once a message has arrived at each node from all of its neighbors, the marginal probability of a node is obtained as follows:

$$p(x_i) \propto \prod_{s \in \mathcal{N}(i)} \mu_{si}(x_i). \tag{4.14}$$

Given the definition of $\nu_{is}(x_i)$ in Eq. (4.12), this can also be written as follows:

$$p(x_i) \propto \nu_{is}(x_i)\mu_{si}(x_i), \tag{4.15}$$

for any $s \in \mathcal{N}(i)$. That is, the marginal probability of node i can be obtained by taking the product of the pair of messages flowing along any edge incident on node i.

A sequential implementation of the SUM-PRODUCT algorithm for factor trees is provided in Figure 4.11.

Consider the example shown in Figure 4.6(a). The factor tree representation of this model is shown in Figure 4.12(b). Let us run through the steps of the SUM-PRODUCT algorithm. In the first step, shown in Figure 4.12(c), the only nodes that are able to send messages are the leaf nodes. These leaf nodes are factor nodes, and the product in Eq. (4.13) is a vacuous product, which by convention we set equal to one. Moreover, the sum in Eq. (4.13) is a vacuous sum. Thus, the message that flows in from a leaf node is simply the factor associated with that node: $\mu_{si}(x_i) = \psi^E(x_i)$, for $i \in \mathcal{V}$.

The second stage in the process is also rather uninteresting. As shown in Figure 4.12(d), the variable nodes X_1 and X_3 are able to send messages in this stage. For each node, the product in Eq. (4.12) is composed of only a single factor, and thus this factor is simply passed along the chain.

Now consider the third stage, shown in Figure 4.12(e). At the factor nodes along the backbone of the chain, a sum is taken over the product of the incoming message and the factor residing at that node. In the case of the message $\mu_{d2}(x_2)$, this yields $\mu_{d2}(x_2) = \sum_{x_1} \psi^E(x_1)\psi(x_1, x_2)$, and, similarly, $\mu_{e2}(x_2) = \sum_{x_3} \psi^E(x_3)\psi(x_2, x_3)$. Note that these messages are the same as the corresponding messages that would pass in a run of the SUM-PRODUCT algorithm for the undirected graph in Figure 4.12(a). That is, we have: $\mu_{d2}(x_2) = m_{12}(x_2)$, and $\mu_{e2}(x_2) = m_{32}(x_2)$.

Finally, in Figure 4.12(f), Figure 4.12(g), and Figure 4.12(h), we show the remaining steps of the algorithm. The reader can again verify a correspondence with the messages that would be computed in Figure 4.12(a): $\mu_{d1}(x_1) = m_{21}(x_1)$ and $\mu_{e3}(x_3) = m_{23}(x_3)$. By the end of the algorithm, a message has passed in both directions along every edge.

In general, if we start with a graph that is an undirected tree and convert to a factor graph, then we find that there is a direct relationship between the "m messages" of the SUM-PRODUCT algorithm for the undirected graph and the "μ messages" of the SUM-PRODUCT algorithm for the factor graph. Consider the graph fragment shown in Figure 4.13(a) and the corresponding factor graph representation in Figure 4.13(b). We claim that $m_{ji}(x_i)$ in the undirected graph is equal to $\mu_{si}(x_i)$ in the factor graph. Indeed, we have:

$$\mu_{si}(x_i) = \sum_{x_{\mathcal{N}(s)\setminus i}} \left(f_s(x_{\mathcal{N}(s)}) \prod_{j \in \mathcal{N}(s)\setminus i} \nu_{js}(x_j) \right) \tag{4.16}$$

$$= \sum_{x_j} \psi(x_i, x_j)\nu_{js}(x_j) \tag{4.17}$$

$$= \sum_{x_j} \psi(x_i, x_j) \prod_{t \in \mathcal{N}(j)\setminus s} \mu_{tj}(x_j) \tag{4.18}$$

$$= \sum_{x_j} \left(\psi^E(x_j)\psi(x_i, x_j) \prod_{t \in \mathcal{N}'(j)\setminus s} \mu_{tj}(x_j) \right), \tag{4.19}$$

where $\mathcal{N}'(j)$ denotes the neighborhood of j, omitting the singleton factor node associated with $\psi^E(x_j)$. We see that the expression for $\mu_{si}(x_i)$ is formally identical to the update equation for $m_{ji}(x_i)$ in Eq. (4.7).

```
SUM-PRODUCT(T, E)
    EVIDENCE(E)
    f = CHOOSEROOT(V)
    for s ∈ N(f)
        μ-COLLECT(f, s)
    for s ∈ N(f)
        ν-DISTRIBUTE(f, s)
    for i ∈ V
        COMPUTEMARGINAL(i)

μ-COLLECT(i, s)
    for j ∈ N(s)\i
        ν-COLLECT(s, j)
    μ-SENDMESSAGE(s, i)

ν-COLLECT(s, i)
    for t ∈ N(i)\s
        μ-COLLECT(i, t)
    ν-SENDMESSAGE(i, s)

μ-DISTRIBUTE(s, i)
    μ-SENDMESSAGE(s, i)
    for t ∈ N(i)\s
        ν-DISTRIBUTE(i, t)

ν-DISTRIBUTE(i, s)
    ν-SENDMESSAGE(i, s)
    for j ∈ N(s)\i
        μ-DISTRIBUTE(s, j)

μ-SENDMESSAGE(s, i)
```

$$\mu_{si}(x_i) = \sum_{x_{\mathcal{N}(s)\setminus i}} \left(f_s(x_{\mathcal{N}(s)}) \prod_{j \in \mathcal{N}(s)\setminus i} \nu_{js}(x_j)\right)$$

```
ν-SENDMESSAGE(i, s)
```

$$\nu_{is}(x_i) = \prod_{t \in \mathcal{N}(i)\setminus s} \mu_{ti}(x_i)$$

```
COMPUTEMARGINAL(i)
```

$$p(x_i) \propto \nu_{is}(x_i)\mu_{si}(x_i)$$

Figure 4.11: A sequential implementation of the SUM-PRODUCT algorithm for a factor tree $\mathcal{T}(\mathcal{V}, \mathcal{F}, \mathcal{E})$. The algorithm works for any choice of root node, and thus we have left CHOOSE-ROOT unspecified. The subroutine EVIDENCE(E) is presented in Figure 4.5.

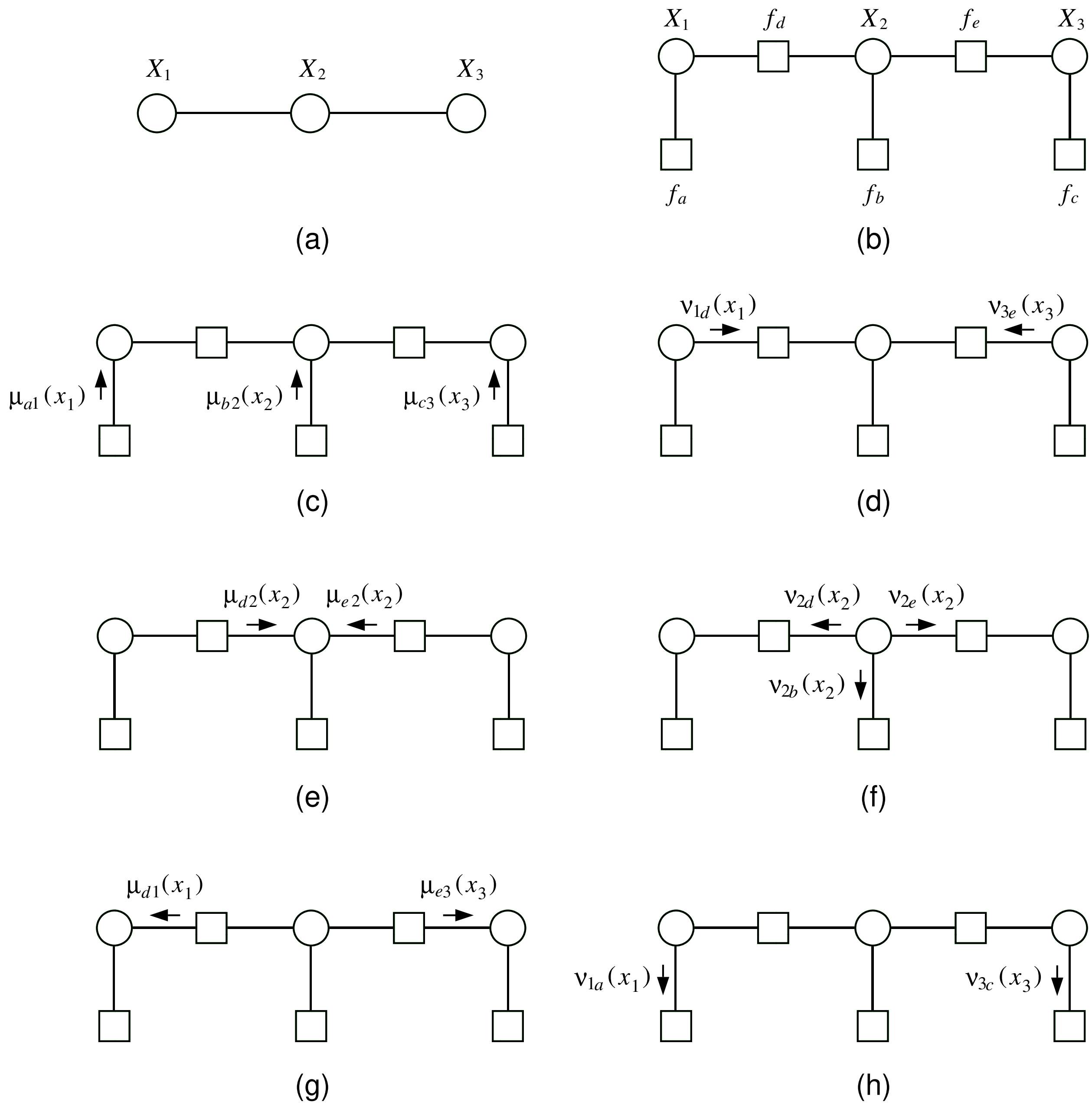


Figure 4.12: (a) A three-node undirected graphical model. (b) The factor tree representation. (c)-(h) A run of the SUM-PRODUCT algorithm on the factor tree.

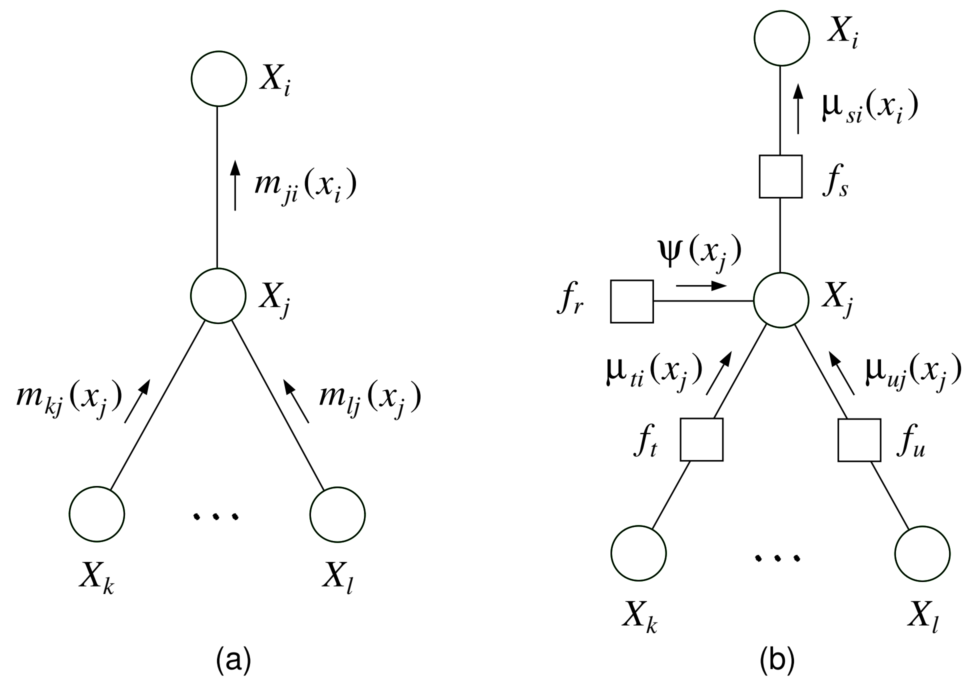


Figure 4.13: (a) A fragment of an undirected tree. (b) The corresponding fragment of a factor tree.

From this observation and an induction argument, it is not difficult to prove that the SUM-PRODUCT algorithm for factor trees is correct for factor trees that are obtained from undirected trees, by simply translating between the two versions of the SUM-PRODUCT algorithm. We leave this as an exercise (Exercise **??**). It is also straightforward to develop a standalone proof by induction that the general SUM-PRODUCT algorithm for factor trees is correct, which we again leave as an exercise.

If a graph is originally a tree (undirected or directed), there is little to be gained by translating to the factor graph framework. The payoff for factor graphs arises when we consider various "tree-like" graphs, to which we now turn.

4.2.3 Tree-like graphs

Consider the graph shown in Figure 4.14(a). Assuming that the three-node cluster in the center of the graph is parameterized by a general non-factorized potential function, the probability distribution associated with the graph is given by:

$$p(x) \propto \psi(x_1, x_2)\psi(x_3, x_5)\psi(x_4, x_6)\psi(x_2, x_3, x_4), \tag{4.20}$$

where for simplicity we have neglected the singleton potentials. Although this graph is not a tree, it is "nearly" a tree. In particular, we could replace the three variable X_2, X_3, and X_4 with a new "super-variable" Z, whose range is the Cartesian product of the ranges of the three individual

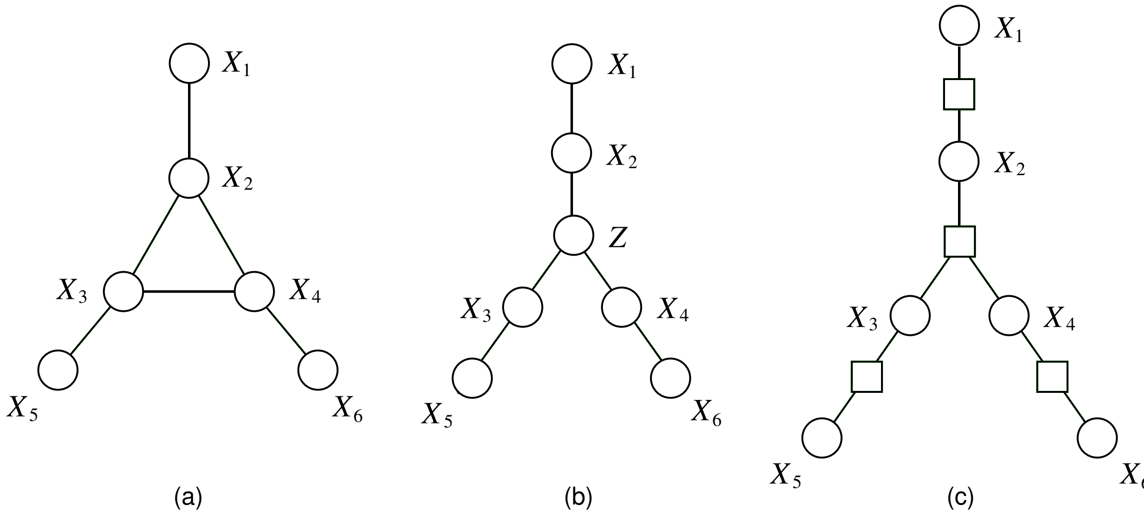


Figure 4.14: (a) An undirected graphical model in which the center cluster of nodes is assumed to be parameterized as a non-factorized potential, $\psi(x_2, x_3, x_4)$. (b) An equivalent undirected model based on the "super-variable" Z. (c) An equivalent factor graph.

variables. By creating new potential functions, $\psi(x_1, z)$, $\psi(x_5, z)$, $\psi(x_6, z)$, and $\psi(z)$, we can mimic the factorization in Eq. (4.20). Moreover, the corresponding undirected graphical model, shown in Figure 4.14(b), is a tree.

We can also capture the probability distribution in Eq. (4.20) using a factor graph. In particular, the graph translates directly to the factor graph shown in Figure 4.14(c). Note that the factor node at the center of the graph has three neighbors—representing the dependency structure of the potential $\psi(x_2, x_3, x_4)$. Note also that the factor graph is a factor tree.

We see that the distribution represented by the tree-like undirected graph in Figure 4.14(a) translates directly to a tree in the factor graph framework. There is no need to invent new variables and new potential functions.

Finally, of most significance is that the SUM-PRODUCT algorithm for factor trees applies directly to the graph in Figure 4.14(c). The fact that the original graph is not a tree is irrelevant—the factor graph *is* a tree, and the algorithm is correct for general factor trees.

In general, if the variables in an undirected graphical model can be clustered into non-overlapping cliques, and the parameterization of each clique is a general, non-factorized potential, then the corresponding factor graph is a tree, and the SUM-PRODUCT applies directly.

4.2.4 Polytrees

A polytree is a tree-like graph that is important enough to merit its own section. In this section we discuss the SUM-PRODUCT algorithm for polytrees, again exploiting the factor graph framework.

As we have discussed, directed trees are essentially equivalent to undirected trees, providing no additional representational capability and no new issues for inference. On the other hand, the

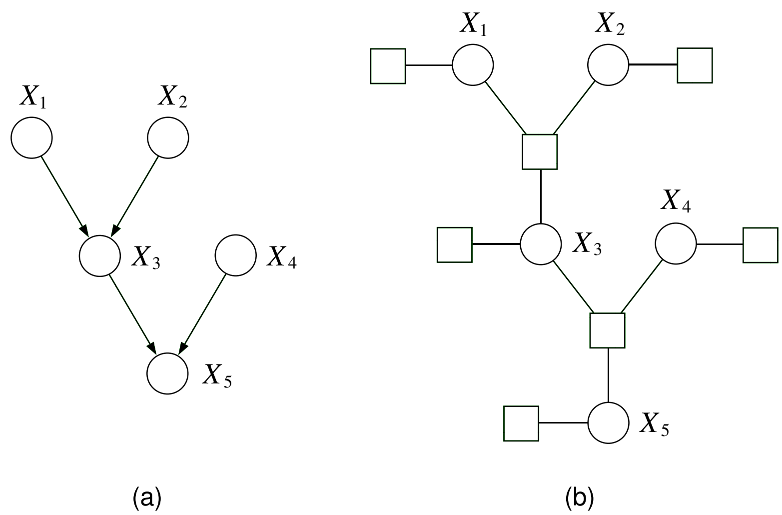


Figure 4.15: (a) A polytree. (b) The factor graph representation of the polytree in (a). Note that the factor graph is a factor tree.

directed graph shown in Figure 4.15(a) is a tree-like graph that does present new capabilities and new issues. As we saw in Chapter 2, the presence of nodes with multiple parents in a directed graph implies a conditional independence semantics that is not available in undirected graphs, including the "explaining-away" semantics that we studied in Chapter 2. Not surprisingly, this semantics has implications for inference, concretely via the conditional probability $p(x_i \mid x_{\pi_i})$ that links a node with its parents.

A *polytree* is a directed graph that reduces to an undirected tree if we convert each directed edge to an undirected edge. Thus, polytrees have no loops in their underlying undirected graph.

One way to treat polytrees is via the "super-variable" approach. That is, we create a new variable for each combination of a node and its parents (each family) and link the super-variables (with undirected edges). It is easy to see that the resulting graph is a tree. This approach, however, suffers from the inelegance alluded to in the previous section.

Alternatively, we can use factor graphs. In Figure 4.15(b), we show the factor graph corresponding to the polytree in Figure 4.15(a). We see that the factor graph is a tree. Moreover, there is a factor corresponding to each family, representing the conditional probability $p(x_i \mid x_{\pi_i})$.

The fact that the factor graph corresponding to a polytree is a tree implies that the Sum-Product algorithm for factor graphs applies directly to polytrees.

Historically, polytrees were an important step along the way in the development of general exact inference algorithms for graphical models. In 1983, Kim and Pearl described a general sum-product-like algorithm for polytrees. As in the case of the Sum-Product algorithm for factor graphs, this algorithm also involves two kinds of messages—"λ messages" flowing from children to parents, and "π messages" flowing from parents to children. The algorithm can be derived readily from the Sum-Product algorithm for the corresponding factor graph. We present the algorithm in Exercise **??**, and ask the reader to provide the derivation.

4.3 Maximum a posteriori probabilities

In this section we discuss a new problem—that of computing *maximum a posteriori probabilities*. Whereas the marginalization problem that we have addressed up until now involves summing over all configurations of sets of random variables, the maximum a posteriori (MAP) problem involves maximizing over such configurations. The problem has two aspects—that of finding the maximal probability and that of finding a configuration that achieves the maximal probability. We begin by focusing on the former problem.[6]

Given a probability distribution $p(x)$, where $x = (x_1, x_2, \ldots, x_n)$, given a partition (E, F) of the indices, and given a fixed configuration $\bar{x}_E$, we wish to compute the maximum a posteriori probability $\max_{x_F} p(x_F \mid \bar{x}_E)$. Although we use the language of "maximum a posteriori probability" to describe this problem, the conditioning turns out to play little significant role in the problem. Indeed:

$$\begin{aligned} \max_{x_F} p(x_F \mid \bar{x}_E) &= \max_{x_F} p(x_F, \bar{x}_E) &\quad (4.21)\\ &= \max_{x} p(x)\delta(x_E, \bar{x}_E) &\quad (4.22)\\ &\triangleq \max_{x} p^E(x), &\quad (4.23) \end{aligned}$$

where $p^E(x)$ is the unnormalized representation of conditional probability introduced in Section 3.1.1. We see that without loss of generality we can study the unconditional case. That is, we treat the general problem of maximizing a nonnegative, factorized function of n variables; this includes as a special case the problem of maximizing such a function when some of the variables are held fixed.

It is important to be clear that the MAP problem is quite distinct from the marginalization problem. Naively, one might think that one could solve the MAP problem by first computing the marginal probability for each variable, and then computing the assignment of each variable that maximizes its individual marginal, but this is incorrect. Consider the pair of variables shown in Figure 4.16. The marginal probability of X is maximized by choosing $X = 1$, and the marginal probability of Y is maximized by choosing $Y = 1$. However, the joint probability of the configuration $(X = 1, Y = 1)$ is equal to zero! The maximizing assignment is $(X = 1, Y = 2)$, which has probability 0.36.

Although the MAP problem is distinct from the marginalization problem, its algorithmic solution is quite similar. To see this, let us return to the example shown in Figure 4.17, a directed graphical model with the following factorization:

$$p(x) = p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1)p(x_4 \mid x_2)p(x_5 \mid x_3)p(x_6 \mid x_2, x_5). \quad (4.24)$$

To solve the MAP problem we expand the maximization into component-wise maximizations, and compute:

$$\max_x p(x) \quad = \quad \max_{x_1} \max_{x_2} \max_{x_3} \max_{x_4} \max_{x_5} \max_{x_6} p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1)p(x_4 \mid x_2)p(x_5 \mid x_3)p(x_6 \mid x_2, x_5)$$

[6]There are generalizations of the MAP problem that involve finding a small set of configurations that have high probability, and finding multiple configurations that have maximal probability when the maximum is not unique. In the current section, we restrict ourselves to the simpler problem of finding a single maximum.

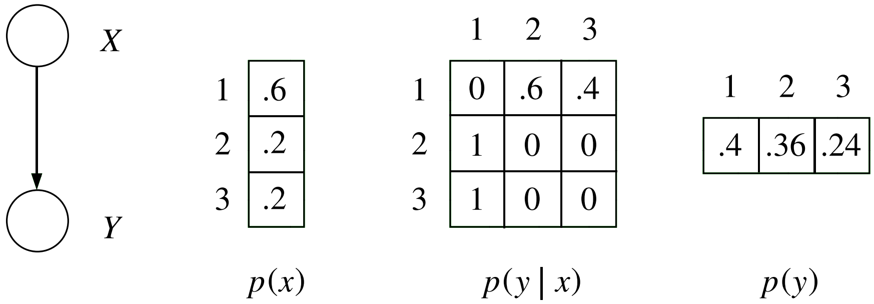


Figure 4.16: The marginal and conditional probabilities for a pair of variables (X, Y). The maximizing values of the individual marginals are $X = 1$ and $Y = 1$, but the configuration $(X = 1, Y = 1)$ has zero probability.

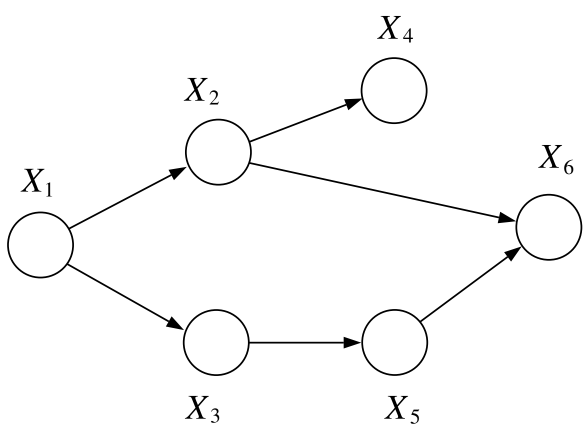


Figure 4.17: A directed graphical model.

MAP-ELIMINATE($\mathcal{G}, E$)
  INITIALIZE($\mathcal{G}$)
  EVIDENCE(E)
  UPDATE($\mathcal{G}$)
  MAXIMUM

INITIALIZE($\mathcal{G}$)
  choose an ordering I
  for each node X_i in $\mathcal{V}$
    place $p(x_i \mid x_{\pi_i})$ on the active list

EVIDENCE(E)
  for each i in E
    place $\delta(x_i, \bar{x}_i)$ on the active list

UPDATE($\mathcal{G}$)
  for each i in I
    find all potentials from the active list that reference x_i and remove them from the active list
    let $\phi_i^{\max}(x_{T_i})$ denote the product of these potentials
    let $m_i^{\max}(x_{S_i}) = \max_{x_i} \phi_i^{\max}(x_{T_i})$
    place $m_i^{\max}(x_{S_i})$ on the active list

MAXIMUM
  $\max_x p^E(x) =$ the scalar value on the active list

Figure 4.18: The MAP-ELIMINATE algorithm for solving the maximum a posteriori problem. Note that after the final node has been eliminated in UPDATE, the active list contains a single scalar value, which is the value returned as the maximum by the algorithm.

$$= \max_{x_1} p(x_1) \max_{x_2} p(x_2 \mid x_1) \max_{x_3} p(x_3 \mid x_1) \max_{x_4} p(x_4 \mid x_2) \max_{x_5} p(x_5 \mid x_3) \max_{x_6} p(x_6 \mid x_2, x_5).$$

These steps should look familiar from our earlier example of marginalization in this graph. Continuing the computation, we perform the maximization with respect to x_6, thereby defining an "intermediate factor" that is a function of x_2 and x_5. Subsequent steps are identical to those of a marginalization computation, with the "sum" operator replaced by the "max" operator.

More generally, all of the derivations that we have presented in this chapter and the previous chapter go through if the "sum" operator is replaced everywhere by the "max" operator. In particular, by making this substitution in ELIMINATE, we obtain a MAP version of ELIMINATE, which we present in Figure 4.18.

The reason that the derivations go through when "sum" is replaced by "max" is that both the "sum-product" pair and the "max-product" pair are examples of an algebraic structure known as a *commutative semiring*. A commutative semiring is a set endowed with two operations—generically referred to as "addition" and "multiplication"—that obey certain laws. In particular, addition and multiplication are both required to be *associative* and *commutative.* Moreover, multiplication is *distributive* over addition:

$$a \cdot b + a \cdot c = a \cdot (b + c). \tag{4.25}$$

This distributive law played a key role in our derivation of ELIMINATE, in which the "sum" operator repeatedly migrates across the "product" operator. Also, the ability to group and reorder intermediate factors was required in the derivation of the ELIMINATE algorithm. In fact, it can be verified that the associative, commutative and distributive laws are all that are needed to derive the ELIMINATE algorithm and the SUM-PRODUCT algorithm. (Note in particular that we do not require division, an operation that is available in the more restrictive algebraic object known as a *ring.*)

If we let the "max" operator play the role of addition, the fact that "max" distributes over "product":

$$\max(a \cdot b, a \cdot c) = a \cdot \max(b, c) \tag{4.26}$$

shows that "max-product" is a semiring (given the easy verification that "max" is associative and commutative), and justifies the MAP-ELIMINATE algorithm in Figure 4.18.

A practical problem with the MAP-ELIMINATE algorithm shown in Figure 4.18 is that the products of probabilities tend to underflow. This can be handled by transforming to the log scale, making use of the fact that:

$$\max_x p^E(x) = \max_x \log p^E(x), \tag{4.27}$$

which holds because the logarithm is a monotone function. Given that the logarithm of a product becomes a sum of logarithms, we see that such an implementation essentially involves working with a "max-sum" pair instead of a "max-product" pair. Fortunately, "max-sum" is also a semiring, in which "max" plays the role of addition and "sum" plays the role of multiplication. Indeed, the distributive law is easily verified:

$$\max(a + b, a + c) = a + \max(b, c), \tag{4.28}$$

as are the associative and commutative laws. Thus we can implement MAP-ELIMINATE algorithm by working with logarithms of potentials, and replacing "product" with "sum."

There are many other commutative semirings, including semirings on polynomials and distributive lattices. We explore some of these commutative semirings in the exercises. The generic ELIMINATE algorithm can be easily adapted to each of these commutative semirings.

In Section **??** we showed that in the case of trees, the ELIMINATE algorithm can be equivalently expressed in terms of a coupled set of equations, or "messages," a line of argument that led to the SUM-PRODUCT algorithm for inference on trees. The same arguments apply to arbitrary commutative semirings, and in particular we can obtain a "MAX-PRODUCT" version of the algorithm as follows:

$$m_{ji}^{\max}(x_i) \quad = \quad \max_{x_j} \left(\psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}^{\max}(x_j) \right) \tag{4.29}$$

$$\max_x p^E(x) \quad = \quad \max_{x_i} \left(\psi^E(x_i) \prod_{j \in \mathcal{N}(i)} m_{ji}^{\max}(x_i) \right). \tag{4.30}$$

Implementing a depth-first traversal of the tree, thereby passing messages from the leaves toward an arbitrarily-defined root, we invoke Eq. (4.30) at the root and obtain the MAP solution.

Is there any value in considering a full message-passing algorithm in which we also send messages from the root back toward the leaves? If the problem is simply that of finding the maximal value of the MAP probability, $\max_x p^E(x)$, then the answer is no. Invoking Eq. (4.30) at multiple nodes in the graph, we obtain exactly the same solution—in all cases we have maximized over all nodes in the graph. However, if our goal is also that of obtaining a maximizing configuration—a configuration x^* such that $x^* \in \arg\max_x p^E(x)$—then we can make use of an appropriately defined outward phase. We explore this issue in the following section.

4.3.1 Maximum a posteriori configurations

Let us now consider the problem of finding a configuration x^* such that $x^* \in \arg\max_x p^E(x)$. This problem can be solved by keeping track of the maximizing values of variables in the inward pass of the MAX-PRODUCT algorithm, and using these values as indices in an outward pass.

Throughout this section we assume that an arbitrary root node f has been chosen, and refer to an "inward pass" in which messages flow from the leaves toward the root, and an "outward pass" in which messages flow from the root toward the leaves.

Note that when the MAX-PRODUCT algorithm arrives at the root node at the end of the inward pass, the final maximization in Eq. (4.30) provides us with a value of the root node that belongs to a maximizing configuration. Thus, letting f denote the root, we compute:

$$x_f^* \in \arg\max_{x_f} \left(\psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}^{\max}(x_f) \right), \tag{4.31}$$

and thereby obtain a value x_f^* that necessarily belongs to a maximizing configuration. Moreover, in principle we could perform an outward pass in which we evaluate Eq. (4.29) for each node from the root to the leaves, and subsequently perform the maximization in Eq. (4.30) at each node. This would yield values x_i^* that belong to maximizing configurations. Unfortunately, however, there is no guarantee that these values all belong to the *same* maximizing configuration. To find a single maximizing configuration we have to work a bit harder.

Suppose that during the inward pass we maintain a record of the maximizing values of nodes when we compute the messages $m_{ji}^{\max}(x_i)$. That is, whenever we send a message $m_{ji}^{\max}(x_i)$ from node j to its parent node i, we also record the maximizing values in a table $\delta_{ji}(x_i)$:

$$\delta_{ji}(x_i) \in \arg\max_{x_j} \left(\psi^E(x_j)\psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}^{\max}(x_j) \right). \tag{4.32}$$

Thus, for each x_i, the function $\delta_{ji}(x_i)$ picks out a value of x_j (there may be several) that achieves the maximum.

Having defined the function $\delta_{ji}(x_i)$ during the inward pass, we use $\delta_{ji}(x_i)$ to define a consistent maximizing configuration during an outward pass. Thus, starting at the root f, we choose a maximizing value x_f^*. Given this value, which we pass to the children of f, we set $x_e^* = \delta_{ef}(x_f^*)$ for each $e \in \mathcal{N}(f)$. This procedure continues outward to the leaves.

The resulting algorithm is summarized in Figure 4.19. Note that the computation of the $m_{ji}^{\max}(x_i)$ messages in the inward pass of this algorithm is identical to the MAP-ELIMINATE algorithm (for undirected trees).

4.4 Conclusions

In this chapter we have presented a basic treatment of algorithms for computing probabilities on graphs. Restricting ourselves to trees, we presented the SUM-PRODUCT algorithm, an algorithm for computing all singleton marginal probabilities. We also presented a SUM-PRODUCT algorithm for factor trees, and showed how this algorithm allows us to compute marginal probabilities for various tree-like graphs, including polytrees. Finally, we showed that the algebra underlying the SUM-PRODUCT algorithm can be abstracted, yielding a general family of propagation algorithms based on commutative semirings. In particular, we presented the MAX-PRODUCT algorithm, an algorithm for computing maximum a posteriori probabilities.

Henceforth we will refer to all such propagation algorithms as *probability propagation algorithms*. While we have restricted ourselves to trees in the current chapter, we will be considering probability propagation algorithms on more general graphs in later chapters.

Thus far we have focused on the problems of representation and inference in graphical models. We return to these problems in Chapters 16 and 17, providing a more general and more formal treatment of topics such as conditional independence and probability propagation. In the intervening chapters, however, we shift to a different line of inquiry. In particular, we now begin to address the problem of interfacing graphical models to data, and we begin to develop methods for evaluating and improving models on the basis of such data. We thus take up the statistical side of the story.

4.5 Historical remarks and bibliography

Max-Product($\mathcal{T}$, E)
    Evidence(E)
    $f =$ ChooseRoot($\mathcal{V}$)
    for $e \in \mathcal{N}(f)$
        Collect(f, e)
    $MAP = \max_{x_f}(\psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m^{\max}_{ef}(x_f))$
    $x^*_f = \arg\max_{x_f}(\psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m^{\max}_{ef}(x_f))$
    for $e \in \mathcal{N}(f)$
        Distribute(f, e)

Collect(i, j)
    for $k \in \mathcal{N}(j) \backslash i$
        Collect(j, k)
    SendMessage(j, i)

Distribute(i, j)
    SetValue(i, j)
    for $k \in \mathcal{N}(j) \backslash i$
        Distribute(j, k)

SendMessage(j, i)

$$m^{\max}_{ji}(x_i) = \max_{x_j}(\psi^E(x_j)\psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \backslash i} m^{\max}_{kj}(x_j))$$

$$\delta_{ji}(x_i) \in \arg\max_{x_j}(\psi^E(x_j)\psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \backslash i} m^{\max}_{kj}(x_j))$$

SetValue(i, j)
    $x^*_j = \delta_{ji}(x^*_i)$

Figure 4.19: A sequential implementation of the Max-Product algorithm for a tree $\mathcal{T}(\mathcal{V}, \mathcal{E})$. The algorithm works for any choice of root node, and thus we have left ChooseRoot unspecified. The subroutine Evidence(E) is presented in Figure 4.5.