Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.438 Algorithms for Inference
Fall 2014

8 Inferences on trees: sum-product algorithm

Recall the two fundamental inference problems in graphical models:

1. *marginalization*, i.e. computing the marginal distribution $p_{\mathsf{x}_i}(x_i)$ for some variable x_i
2. *mode*, or *maximum a posteriori (MAP)* estimation, i.e. finding the most likely joint assignment to the full set of variables $\mathbf{x}$.

In the last lecture, we introduced the elimination algorithm for performing marginalization in graphical models. We saw that the elimination algorithm always produces an exact solution and can be applied to any graph structure. Furthermore, it is not entirely naive, in that it makes use of the graph structure to save computation. For instance, recall that in the grid graph, its computational complexity grows as $O\left(|\mathcal{X}|^{\sqrt{N}}\right)$, while the naive brute force computation requires $O\left(|\mathcal{X}|^N\right)$ operations.

However, if we wish to retrieve the marginal distributions for multiple different variables, we would need to run the elimination algorithm from scratch each time. This would entail much redundant computation. It turns out that we can recast the computations in the elimination algorithm as messages passed between nodes in the network, and that these messages can be reused between different marginalization queries. The resulting algorithm is known as the *sum-product algorithm*. In this lecture, we look at the special case of sum-product on trees. In later lectures, we will extend the algorithm to graphs in general. We will also see that a similar algorithm can be applied to obtain the MAP estimate.

8.1 Elimination algorithm on trees

Recall that a graph $\mathcal{G}$ is a *tree* if any two nodes are connected by exactly one path. This definition implies that all tree graphs have exactly $N-1$ edges and no cycles.

Throughout this lecture, we will use the recurring example of the tree graphical model shown in Figure 1. Suppose we wish to obtain the marginal distribution p_{x_1} using the elimination algorithm. The efficiency of the algorithm depends on the elimination ordering chosen. For instance, if we choose the suboptimal ordering $(2,4,5,3,1)$, the elimination algorithm produces the sequence of graph structures shown in Figure 2. After the first step, all of the neighbors of x_2 are connected, resulting in a table of size $|\mathcal{X}|^3$. For a more drastic example of a suboptimal ordering, consider the star graph shown in Figure 3. If we eliminate x_1 first, the resulting graph is a fully connected graph over the remaining variables, resulting in a table of size $|\mathcal{X}|^{N-1}$. This is clearly undesirable.

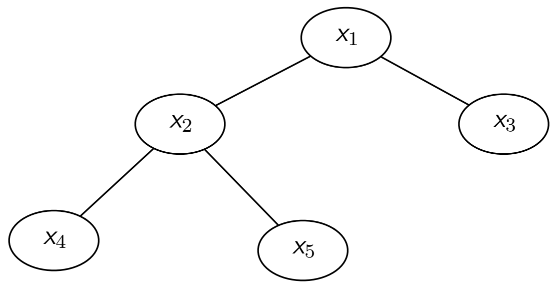

Figure 1: A tree-structured graphical model which serves as a recurring example throughout this lecture.

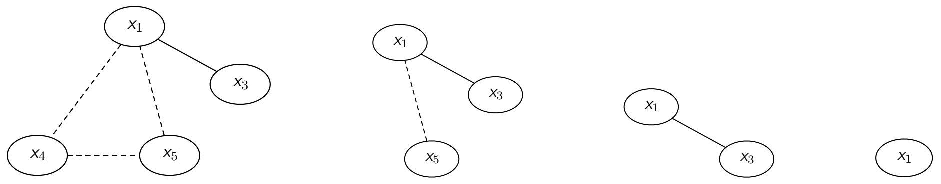

Figure 2: The sequence of graph structures obtained from the elimination algorithm on the graph from Figure 1 using the suboptimal ordering $(2, 4, 5, 3, 1)$.

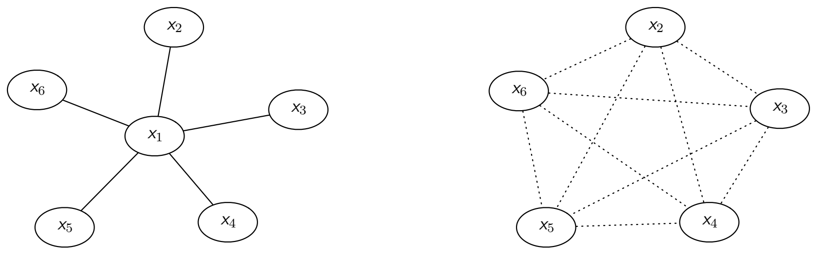

Figure 3: A star-shaped graph and the resulting graph after eliminating variable x_1.

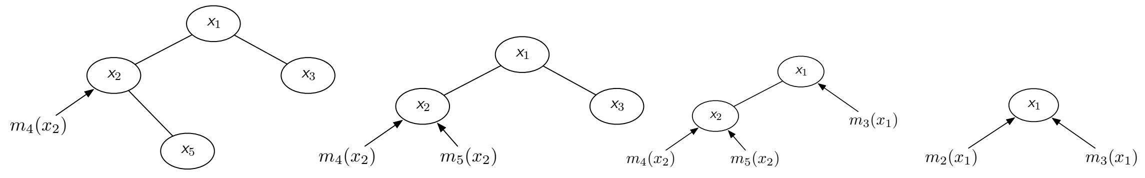

Figure 4: The sequence of graph structures and messages obtained from the elimination algorithm on the graph from Figure 1 using an optimal ordering $(4, 5, 3, 2, 1)$.

Fortunately, for tree graphs, it is easy to find an ordering which adds no edges. Recall that, in last lecture, we saw that starting from the edges of a graph is a good heuristic. In the case of trees, "edges" correspond to leaf nodes, which suggests starting from the leaves. In fact, each time we eliminate a variable, the graph remains a tree, so we can choose an ordering by iteratively removing a leaf node. (Note that the root node must come last in the ordering; however, it is easy to show that every tree has at least two leaves.) In the case of the graph from Figure 1, one such ordering is $(4, 5, 3, 2, 1)$. The resulting graphs and messages are shown in Figure 4.

In a tree graph, the maximal cliques are exactly the edges. Therefore, by the Hammersley-Clifford Theorem, if we assume the joint distribution is strictly positive, we can represent it (up to normalization) as a product of potentials $\psi_{ij}(x_i, x_j)$ for each edge (i, j). However, it is often convenient to include unary potentials as well, so we will assume a redundant representation with unary potentials $\phi_i(x_i)$ for each variable x_i. In other words, we assume the factorization

$$p_{\mathbf{x}}(\mathbf{x}) = \frac{1}{Z} \prod_{i \in \mathcal{V}} \phi_i(x_i) \prod_{(i,j) \in \mathcal{E}} \psi_{ij}(x_i, x_j). \tag{1}$$

The messages produced in the course of the algorithm are:

$$\begin{aligned}
m_4(x_2) &= \sum_{x_4} \phi_4(x_4)\psi_{24}(x_2, x_4) \\
m_5(x_2) &= \sum_{x_5} \phi_5(x_5)\psi_{25}(x_2, x_5) \\
m_3(x_1) &= \sum_{x_3} \phi_3(x_3)\psi_{13}(x_1, x_3) \\
m_2(x_1) &= \sum_{x_2} \phi_2(x_2)\psi_{12}(x_1, x_2)m_4(x_2)m_5(x_2) \qquad (2)
\end{aligned}$$

Finally, we obtain the marginal distribution over x_1 by multiplying the incoming messages with its unary potential, and then marginalizing. In particular,

$$p_{\mathsf{x}_1}(x_1) \propto \phi_1(x_1)m_2(x_1)m_3(x_1). \tag{3}$$

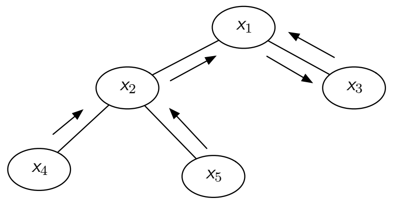

Figure 5: All messages required to compute the marginals over $p_{x_1}(x_1)$ and $p_{x_3}(x_3)$ for the graph in Figure 1.

Now consider the computational complexity of this algorithm. Each of the messages produced has $|\mathcal{X}|$ values, and computing each value requires summing over $|\mathcal{X}|$ terms. Since this must be done for each of the $N-1$ edges in the graph, the total complexity is $O(N|\mathcal{X}|^2)$, i.e. linear in the graph size and quadratic in the alphabet size.[1]

8.2 Sum-product algorithm on trees

Returning to Figure 1, suppose we want to compute the marginal for another variable x_3. If we use the elimination ordering (5, 4, 2, 1, 3), the resulting messages are:

$$\begin{aligned} m_5(x_2) &= \sum_{x_5} \phi_5(x_5)\psi_{25}(x_2, x_5) \\ m_4(x_2) &= \sum_{x_4} \phi_4(x_4)\psi_{24}(x_2, x_4) \\ m_2(x_1) &= \sum_{x_2} \phi_2(x_2)\psi_{12}(x_1, x_2)m_4(x_2)m_5(x_2) \\ m_1(x_3) &= \sum_{x_1} \phi_1(x_1)\psi_{13}(x_1, x_3)m_2(x_1) \end{aligned} \tag{4}$$

Notice that the first three messages m_1, m_4, and m_2 are all strictly identical to the corresponding messages from the previous computation. The only new message to be computed is $m_1(x_3)$, as shown in Figure 5.

As this example suggests, we can obtain the marginals for every node in the graph by computing $2(N-1)$ messages, one for each direction along each edge. When computing the message $m_{i\to j}(x_j)$, we need the incoming messages $m_{k\to i}(x_i)$ for its other neighbors $k \in N(i) \setminus \{j\}$, as shown in Figure 6. (Note: the $\setminus$ symbol denotes the difference of two sets, and $N(i)$ denotes the neighbors of node i.) Therefore, we need to choose an ordering over messages such that the prerequisites are available at each step. One way to do this is through the following two-step procedure.

[1]Note that this analysis does not include the time for computing the products in each of the messages. A naive implementation would, in fact, have higher computational complexity. However, with the proper bookkeeping, we can reuse computations between messages to ensure that the total complexity is $O(N|\mathcal{X}|^2)$.

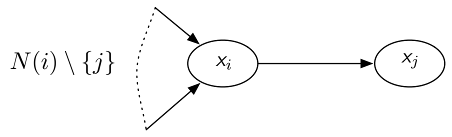

Figure 6: The message $m_i(x_i)$ depends on each of the incoming messages $m_k(x_i)$ for x_i's other neighbors $N(i) \setminus \{j\}$.

- Choose an arbitrary node i as the root, and generate messages going towards it using the elimination algorithm described in Section 8.1.
- Compute the remaining messages, working outwards from the root.

We now combine these insights into the *sum-product algorithm on trees.* Messages are computed in the order given above using the rule:

$$m_{i \to j}(x_j) \;=\; \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i)\setminus\{j\}} m_{k \to i}(x_i). \tag{5}$$

Note that, in order to disambiguate messages sent from node i, we explicitly write $m_{i \to j}(x_j)$ rather than simply $m_i(x_j)$. Then, the marginal for each variable is obtained using the formula:

$$p_{x_i}(x_i) \;\propto\; \phi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i). \tag{6}$$

We note that the sum-product algorithm can also be viewed as a dynamic programming algorithm for computing marginals. This view will become clearer when we discuss hidden Markov models.

8.3 Parallel sum-product

The sum-product algorithm as described in Section 8.2 is inherently sequential: the messages must be computed in sequence to ensure that the prerequisites are available at each step. However, the algorithm was described in terms of simple, local operations corresponding to different variables, which suggests that it might be parallelizable. This intuition turns out to be correct: if the updates (6) are repeatedly applied in parallel, it is possible to show that the messages will eventually converge to their correct values. More precisely, letting $m_i^t(x_j)$ denote the messages at time step t, we apply the following procedure:

1. Initialize all messages $m^0_{i \to j}(x_j) = 1$ for all $(i, j) \in E$.
2. Iteratively apply the update

$$m^{t+1}_{i \to j}(x_j) \;=\; \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i)\setminus\{j\}} m^t_{k \to i}(x_i) \tag{7}$$

Intuitively, this procedure resembles fixed point algorithms for solving equations of the form $f(x) = x$. Fixed point algorithms choose some initial value x_0 and then iteratively apply the update $x^{t+1} = f(x^t)$. We can view (6) not as an update rule, but as a set of equations to be satisfied. The rule (7) can be viewed as a fixed point update for (6). You will prove in a homework exercise that this procedure will converge to the correct messages (6) in d iterations, where d is the diameter of the tree (i.e. the length of the longest path).

Note that this parallel procedure entails significant overhead: each iteration of the algorithm requires computing the messages associated with every edge. We saw in Section 8.1 that this requires $O\left(N|\mathcal{X}|^2\right)$ time. This is the price we pay for parallelism. Parallel sum-product is unlikely to pay off in practice unless the diameter of the tree is small. However, in a later lecture we will see that it naturally leads to *loopy belief propagation*, where the update rule (7) is applied to a graph which isn't a tree.

8.4 Efficient implementation

In our complexity analysis from Section 8.1, we swept under the rug the details of exactly how the messages are computed. Consider the example of the star graph shown in Figure 3, where this time we intelligently choose the center node x_1 as the root. When we compute the outgoing messages $m_{1\to j}(x_j)$, we must first multiply together all the incoming messages $m_{k\to 1}(x_1)$. Since there are $N-2$ of these, the product requires roughly $|\mathcal{X}|N$ computations. This must be done for each of the $N-1$ outgoing messages, so these products contribute approximately $|\mathcal{X}|N^2$ computations in total. This is quadratic in N, which is worse than the linear dependency we stated earlier. More generally, for a tree graph $\mathcal{G}$, these products require $O\left(|\mathcal{X}|\sum_i d_i^2\right)$ time, where d_i is the degree (number of neighbors) of node i.

However, in parallel sum-product, we can share computation between the different messages by computing them simultaneously as follows:

1. Compute
$$\mu_i^t(x_i) \quad = \quad \left(\prod_{k\in N(i)} m_{k\to i}^t(x_i)\right)\phi_i(x_i) \tag{8}$$
2. For all $j \in N(i)$, compute
$$m_{i\to j}^{t+1}(x_j) = \sum_{x_i\in\mathcal{X}} \frac{\psi_i(x_i,x_j)\mu_i^t(x_i)}{m_{j\to i}^t(x_i)} \tag{9}$$

Using this algorithm, each update (8) can be computed in $O(|\mathcal{X}|d_i)$ time, so the cost per iteration is $O\left(|\mathcal{X}|\sum_i d_i\right) = O\left(|\mathcal{X}|N\right)$. Computing (9) still takes $O\left(|\mathcal{X}|^2\right)$ per node, so the overall running time is $O\left(|\mathcal{X}|^2N\right)$ per iteration, or $O\left(|\mathcal{X}|^2Nd\right)$ total. (Recall that d is the diameter of the graph.) A similar strategy can be applied to the sequential algorithm to achieve a running time of $O(|\mathcal{X}|^2N)$.