10-708: Probabilistic Graphical Models 10-708, Spring 2014

Lecture 7: Learning Fully Observed UGMs (Cont'd) & Exact Inference

Lecturer: Matthew Gormley *Scribes: Keith Maki, Anbang Hu, Jining Qin*

1 Learning Fully Observed UGMs (Cont'd)

1.1 Fixed point iteration for optimization

Fixed point iteration is a general tool for solving systems of equations. It is also applicable to optimization. In order to maximize a given objective function $J(\boldsymbol{\theta})$, we can do the following:

1. Compute derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}$ and set it to zero: $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} = 0 = f(\boldsymbol{\theta})$, where we introduce f to denote the derivative.
2. Rearrange the equation such that one of the parameters appear on the LHS: $\theta_i = g(\boldsymbol{\theta})$.
3. Rewrite the equation with timestamp: $\theta_i^{(t+1)} = g(\boldsymbol{\theta}^{(t)})$
4. Initialize the parameters $\boldsymbol{\theta}$.
5. Update parameters and increment t.
6. Repeat step 5 until convergence.

1.2 Iterative proportional fitting (IPF)

In iterative proportional fitting (IPF), our goal is to maximize log-likelihood

$$\ell(\boldsymbol{\theta}; \mathcal{D}) = \sum_{n=1}^{N} \log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \tag{1}$$

where $p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_C \psi_C(\mathbf{x}_C)$. Following the suit of fixed point iteration algorithm, after letting the derivative of Eq.1 equal zero, we get

$$\frac{\partial \ell(\boldsymbol{\theta}; \mathcal{D})}{\partial \psi_C(\mathbf{x}_C)} = \frac{m(\mathbf{x}_C)}{\psi_C(\mathbf{x}_C)} - N \frac{p(\mathbf{x}_C)}{\psi_C(\mathbf{x}_C)} = 0 \tag{2}$$

After rearranging one of the parameters to the LHS in Eq.2 and define $\tilde{p}(\mathbf{x}_C) \triangleq m(\mathbf{x}_C)/N$, we obtain

$$\psi_C(\mathbf{x}_C) = \psi_C(\mathbf{x}_C) \frac{\tilde{p}(\mathbf{x}_C)}{p(\mathbf{x}_C)} \tag{3}$$

So we can set up the update rule:

$$\psi_C^{(t+1)}(\mathbf{x}_C) = \psi_C^{(t)}(\mathbf{x}_C)\frac{\tilde{p}(\mathbf{x}_C)}{p^{(t)}(\mathbf{x}_C)} \tag{4}$$

Loop through each clique C in $\mathcal{C}$ and update each potential function while incrementing t until the potential functions converge to the maximum likelihood estimates.

IPF requires the potentials to be fully parameterized: $\psi_C(\mathbf{x}_C) = \theta_{C,\mathbf{x}_C}$. IPF iterates a set of fixed-point equations Eq.4. IPF is also a coordinate ascent algorithm (coordinates are parameters of clique potentials). IPF guarantees that the log-likelihood Eq.1 increases at each step and converges to a global maximum.

1.3 Generalized iterative scaling (GIS)

General potentials for large cliques are exponentially costly for inference and have exponential numbers of parameters that we must learn from limited data. One solution is to change the graphical model to make cliques smaller. But this changes the dependencies, and may force us to make more independence assumptions than we would like. Another solution is to keep the same graphical model, but use a less general parameterization of the clique potentials. This motivates the feature based potential

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})}\exp\left\{\sum_i \theta_i f_i(\mathbf{x})\right\} \tag{5}$$

which is exponential family potential.

For MRF with feature based potential, we apply generalized iterative scaling (GIS), because it allows more general parameterization and can be seen as IPF extended to other settings. In generalized iterative scaling (GIS), instead of optimizing the log-likelihood function directly, we repeatedly increase a tight lower bound. Given the average log-likelihood function:

$$\tilde{\ell}(\boldsymbol{\theta};\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N}\log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \tag{6}$$

Denoting $\tilde{p}(\mathbf{x}) = \frac{m(\mathbf{x})}{N}$ as before, and substituting Eq.5 into Eq.6 we have

$$\begin{aligned}
\tilde{\ell}(\boldsymbol{\theta};\mathcal{D}) &= \frac{1}{N}\sum_{\mathbf{x}} m(\mathbf{x})\log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \\
&= \sum_{\mathbf{x}}\frac{m(\mathbf{x})}{N}\log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \\
&= \sum_{\mathbf{x}}\tilde{p}(\mathbf{x})\log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \\
&= \sum_{\mathbf{x}}\tilde{p}(\mathbf{x})\sum_i \theta_i f_i(\mathbf{x}) - \log Z(\boldsymbol{\theta})
\end{aligned}$$

We would like to attack the lower bound of the average log-likelihood. Because $\log Z(\boldsymbol{\theta}) \leq \mu Z(\boldsymbol{\theta}) - \log\mu - 1$, $\forall\mu$. We choose $\mu = Z^{-1}(\boldsymbol{\theta}^{(t)})$, then

$$\tilde{\ell}(\boldsymbol{\theta};\mathcal{D}) \geq \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \sum_i \theta_i f_i(\mathbf{x}) - \frac{Z(\boldsymbol{\theta})}{Z(\boldsymbol{\theta}^{(t)})} - \log Z(\boldsymbol{\theta}^{(t)}) + 1$$

Let $\Delta\theta_i^{(t)} = \theta_i - \theta_i^{(t)}$, we can get

$$\begin{aligned}
\tilde{\ell}(\boldsymbol{\theta};\mathcal{D}) &\geq \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \sum_i \theta_i f_i(\mathbf{x}) - \frac{1}{Z(\boldsymbol{\theta}^{(t)})} \sum_{\mathbf{x}} \exp\left\{\sum_i \theta_i f_i(\mathbf{x})\right\} - \log Z(\boldsymbol{\theta}^{(t)}) + 1 \\
&= \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \sum_i \theta_i f_i(\mathbf{x}) - \frac{1}{Z(\boldsymbol{\theta}^{(t)})} \sum_{\mathbf{x}} \exp\left\{\sum_i \theta_i^{(t)} f_i(\mathbf{x})\right\} \exp\left\{\sum_i \Delta\theta_i^{(t)} f_i(\mathbf{x})\right\} - \log Z(\boldsymbol{\theta}^{(t)}) + 1 \\
&= \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \sum_i \theta_i f_i(\mathbf{x}) - \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\theta}^{(t)}) \exp\left\{\sum_i \Delta\theta_i^{(t)} f_i(\mathbf{x})\right\} - \log Z(\boldsymbol{\theta}^{(t)}) + 1
\end{aligned}$$

Due to the convexity of exponential function, we can apply Jensen's inequality ($\exp\left\{\sum_i f_i(\mathbf{x})\Delta\theta_i^{(t)}\right\} \leq \sum_i f_i(\mathbf{x}) \exp\left\{\Delta\theta_i^{(t)}\right\}$) to get

$$\tilde{\ell}(\boldsymbol{\theta};\mathcal{D}) \geq \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \sum_i \theta_i f_i(\mathbf{x}) - \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\theta}^{(t)}) \sum_i f_i(\mathbf{x}) \exp\left\{\Delta\theta_i^{(t)}\right\} - \log Z(\boldsymbol{\theta}^{(t)}) + 1 \tag{7}$$

We denote the RHS of Eq.7 by $\Lambda(\boldsymbol{\theta})$. $\Lambda(\boldsymbol{\theta})$ is the lower bound of which we should take derivative:

$$\frac{\partial\Lambda(\boldsymbol{\theta})}{\partial\theta_i} = \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) f_i(\mathbf{x}) - \exp\left\{\Delta\theta_i^{(t)}\right\} \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\theta}^{(t)}) f_i(\mathbf{x}) \tag{8}$$

Setting Eq.8 to zero, we obtain the fixed point iteration update rule:

$$\Delta\theta_i^{(t)} = \log\left(\frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) f_i(\mathbf{x})}{\sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\theta}^{(t)}) f_i(\mathbf{x})}\right) = \log\left(\frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) f_i(\mathbf{x})}{\sum_{\mathbf{x}} p^{(t)}(\mathbf{x}) f_i(\mathbf{x})} Z(\boldsymbol{\theta}^{(t)})\right) \tag{9}$$

or

$$\theta_i^{(t+1)} = \theta_i^{(t)} + \log\left(\frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) f_i(\mathbf{x})}{\sum_{\mathbf{x}} p^{(t)}(\mathbf{x}) f_i(\mathbf{x})} Z(\boldsymbol{\theta}^{(t)})\right) \tag{10}$$

where $p^{(t)}(\mathbf{x})$ is the unnormalized version of $p(\mathbf{x}|\boldsymbol{\theta}^{(t)})$.

This update rule is applied for each dimension of $\boldsymbol{\theta}$ as t is incremented until convergence to the maximum likelihood estimates for MRF.

Let's take another look at Eq.6:

$$\begin{aligned}\tilde{\ell}(\boldsymbol{\theta}; \mathcal{D}) &= \frac{1}{N}\sum_{n=1}^{N} \log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \\ &= \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}) \\ &= \sum_{C}\sum_{\mathbf{x}_C} \tilde{p}(\mathbf{x}_C) \log \psi_C(\mathbf{x}_C) - \log Z(\boldsymbol{\theta})\end{aligned}$$

Differentiating it, we get

$$\begin{aligned}\frac{\partial \ell(\boldsymbol{\theta}, \mathcal{D})}{\partial \theta_k} &= \left(\sum_{C}\sum_{\mathbf{x}_C} \tilde{p}(\mathbf{x}_C) f_{C,k}(\mathbf{x}_C)\right) - \left(\sum_{C}\sum_{\mathbf{x}_C} p(\mathbf{x}_C) f_{C,k}(\mathbf{x}_C)\right) \\ &= \mathop{\mathbb{E}}_{\mathbf{x}\sim\tilde{p}(\cdot|\mathcal{D})}[f_{\cdot,k}(\mathbf{x})] - \mathop{\mathbb{E}}_{\mathbf{x}\sim p(\cdot|\mathcal{D})}[f_{\cdot,k}(\mathbf{x})]\end{aligned}$$

where $f_{\cdot,k} = \sum_C f_{C,k}(\mathbf{x}_C)$. Here $\tilde{p}(\cdot|\mathcal{D})$ is the empirical distribution of data points $\mathbf{x}$ and $p(\cdot|\mathcal{D})$ is the model distribution.

Note that neither IPF nor GIS works well in practical settings. The reason they are used in learning of conditional random field is mostly historical. People used these models for log-linear models and exponential family models and they naturally tried to adapt them for conditional random field. Later people realized we have a lot of optimization methods available. Using the log-likelihood function as objective, together with its gradient and sometimes Hessian, we can get the maximum likelihood estimate for the model by optimizing the log-likelihood directly. And that corresponds to the gradient-based learning methods.

1.4 Gradient-based learning method

In a gradient-based learning algorithm. We follow the steps below:

1. Write down the objective function.
2. Compute the partial derivatives of the objective (i.e. gradient, and sometimes also Hessian).
3. Feed objective function and derivatives into black box optimization algorithms.
4. Retrieve optimal parameters from black box.

The black box optimization algorithms include:

1. *Newton's method.* It requires the gradient and Hessian of the objective function.
2. *Hessian-free/Quasi-Newton methods.* When the gradient or Hessian is hard to obtain either symbolically or numerically, there are ways to get around it by calculating an approximate partial derivative. They include conjugate gradient method and L-BFGS method, etc.
3. *Stochastic gradient methods.* They work better when the objective function is expressed as the sum of a lot of differentiable functions. They include stochastic gradient descent, stochastic meta-descent, AdaGrad, etc.

1.5 Regularization

When we are learning a model using training data set, we are really trying to find a model that maximizes the probability density of the training data set given a set of parameters. Sometimes that leads to fitting the noise but not signal present in the data set. This phenomenon is called overfitting. Overfitting can greatly undermine the generalizability of a model. Overfitting can be avoided by reasonable regularization.

In order to avoid overfitting in learning with gradient-based methods, we add a penalty term (a.k.a regularization term) to the objective log-likelihood function:

$$J(\boldsymbol{\theta}) = \ell(\boldsymbol{\theta}) + r(\boldsymbol{\theta}) \tag{11}$$

For L2 regularization,

$$r(\boldsymbol{\theta}) = \frac{\lambda}{2}(\|\boldsymbol{\theta}\|_2)^2 = \frac{\lambda}{2}\sum_{k=1}^{K}\theta_k^2 \tag{12}$$

It can be shown L2 regularization is equivalent to MAP estimation with prior $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \frac{1}{\lambda}\mathbf{I})$.

For L1 regularization,

$$r(\boldsymbol{\theta}) = \lambda \|\boldsymbol{\theta}\|_1 = \lambda \sum_{k=1}^{K} |\theta_k| \tag{13}$$

It can be shown L1 regularization is equivalent to MAP estimation with prior $\boldsymbol{\theta} \sim \mathit{Laplace}(\lambda)$. The L1 regularization is more likely to produce sparse parameter estimates so it is often used for model selection.

2 Exact Inference

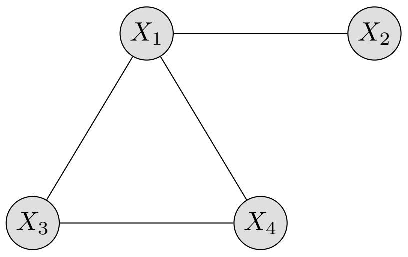
2.1 Factor graphs

Although the techniques discussed so far have focused on either directed or undirected graphical models, these two representations may be unified by a third representation known as a factor graph. Factor graphs unify the representation of Markov Random Fields and Bayes Nets by explicitly encoding relations between the random variables in a GM with arbitrary functions called factors. This representation naturally supports inference techniques which apply to both directed and undirected graphical models, such as the class of belief propagation algorithms introduced in section 2.4 and 2.5 below.

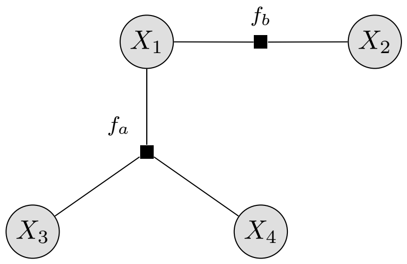
Visually, factor graphs are represented using a bipartite graph, with two types of vertices, representing the random variables and the factors, respectively. Random variables which share a factor are connected to that factor by undirected edges.

In a factor graph, the factors serve as potential tables, assigning weights locally to each possible configuration of the connected variables. Indeed, for undirected graphical models, the factors have a natural one-to-one mapping onto the clique potentials, as shown in Figure 1. For directed graphical models, factors represent the marginal and conditional distributions in the factored joint distribution as shown in Figure 2.

It is important to note also that in addition to unifying the representation between directed and undirected graphical models, the structure of the underlying factor graph may be simplified from that of the original model. In particular, we emphasize the fact that a non-tree graphical model may have a factor graph which is a tree, referred to as a *factor tree*. For example, notice that maximal cliques in the undirected case (e.g. Figure 1) and polytrees in the directed case (e.g. Figure 3) correspond to factor trees.

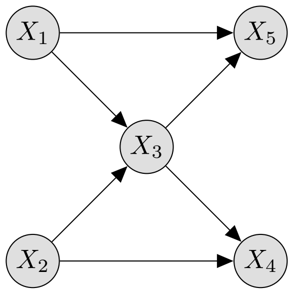


(a) $P(\mathbf{x}) \propto \psi_{134}(x_1, x_3, x_4)\psi_{12}(x_1, x_2)$

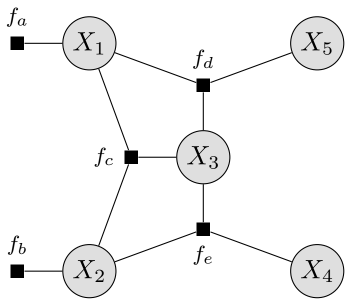


(b) $P(\mathbf{x}) \propto f_a(x_1, x_3, x_4)f_b(x_1, x_2)$

Figure 1: Markov Random Field with equivalent factor graph representation

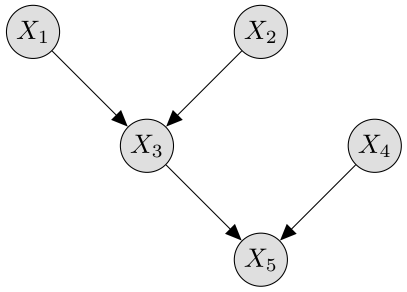


(a) $P(\mathbf{x}) = P(x_1)P(x_2)P(x_3|x_1, x_2)$
$P(x_5|x_1, x_3)P(x_4|x_2, x_3)$

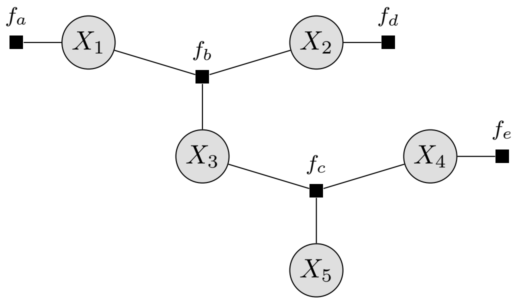


(b) $P(\mathbf{x}) = f_a(x_1)f_b(x_2)f_c(x_3, x_1, x_2)$
$f_d(x_5, x_1, x_3)f_e(x_4, x_2, x_3)$

Figure 2: Bayes net with equivalent factor graph representation



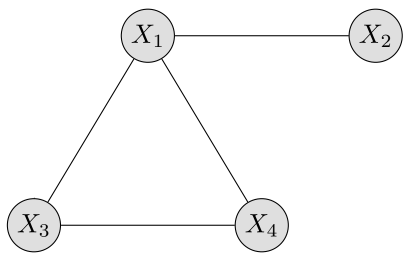
(a) $P(\mathbf{x}) = P(x_1)P(x_3|x_1, x_2)$
$P(x_5|x_3, x_4)P(x_2)P(x_4)$



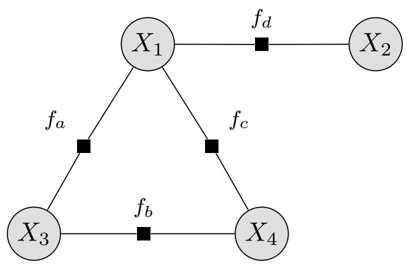
(b) $P(\mathbf{x}) = f_a(x_1)f_b(x_3, x_1, x_2)$
$f_c(x_5, x_3, x_4)f_d(x_2)f_e(x_4)$

Figure 3: Bayes net polytree with corresponding factor tree representation

However, note that for an MRF, there may be more than one factor graph which represent the same graphical model. This is a result of the ambiguity inherent in the clique structure of undirected graphical models. The structure of the factor graph implicitly identifies the set of cliques used to factor the joint distribution of the MRF. As shown in Figures 1 and 4, this choice of factorization may determine whether the factor graph is a factor tree, which can have implications for inference.



(a) $P(\mathbf{x}) \propto \psi_{13}(x_1, x_3)\psi_{34}(x_3, x_4)$
$\psi_{14}(x_1, x_4)\psi_{12}(x_1, x_2)$



(b) $P(\mathbf{x}) \propto f_a(x_1, x_3)f_b(x_3, x_4)$
$f_c(x_1, x_4)f_d(x_1, x_2)$

Figure 4: Markov Random Field with equivalent factor graph representation

2.2 Inference problems

In inference, there are three task, all of which are NP-hard in general case. The three tasks are listed below:

1. **Marginal Inference**. Compute marginals of variables and cliques:

$$p(x_i) = \sum_{\mathbf{x}':x'_i=x_i} p(\mathbf{x}'|\boldsymbol{\theta})$$
$$p(\mathbf{x}_C) = \sum_{\mathbf{x}':\mathbf{x}'_C=\mathbf{x}_C} p(\mathbf{x}'|\boldsymbol{\theta})$$

2. **Partition Function**. Compute the normalization constant:

$$Z(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \prod_{C\in\mathcal{C}} \psi_C(\mathbf{x}_C)$$

3. **MAP Inference**. Compute the variable assignment with highest probability:

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\theta})$$

Computing marginals by sampling on factor graph is NP-hard in general. In practice, we use MCMC to draw an approximate sample fast. In addition, sampling finds the high-probability value x_i efficiently, but it takes too many samples to see a low-probability ones.

2.3 Variable elimination

Variable elimination is a simple and general exact inference algorithm for graphical models. During the following discussion, we omit the parameters $\boldsymbol{\theta}$ for simplicity.

Let $X = \{X_1, \ldots, X_n\}$ denote the set of variable nodes, each of which can range from 1 to k, $F = \{\psi_{\alpha_1}, \ldots, \psi_{\alpha_s}\}$ denote the set of factor nodes, where $\alpha_1, \ldots, \alpha_s \subseteq \{1, \ldots, n\}$. To compute marginal $p(x_1)$, naively, we do the following:

$$p(x_1) = \frac{1}{Z} \sum_{x_2,\ldots,x_n} \prod_{i=1}^{s} \psi_{\alpha_i}(x_{\alpha_i})$$

This involves $O(k^{n-1})$ additions. Instead, we can choose an ordering $\mathcal{I}$ (in which x_1 appears at the end) and capitalize on the factorization of $p(\mathbf{x})$. But first, we need to know several concepts. A *potential* can be a factor ψ_{α_i}, or an "intermediate" table$m_i(x_{S_i})$, where $m_i(x_{S_i})$ results from elimination of variable x_i. An *active list* stores all these potentials. The variable elimination algorithm runs as follows:

1. Initialize active list with all potentials in the factor graph and choose an ordering $\mathcal{I}$ in which x_1 appears last.
2. For each $i \in \mathcal{I}$, do the following
   - find all potentials from the active list that reference x_i and remove them from the active list
   - let $\phi_i(x_{T_i})$ denote the product of these potentials
   - let $m_i(x_{S_i}) = \sum_{x_i} \phi_i(x_{T_i})$
   - place $m_i(x_{S_i})$ on the active list
3. eventually, we get
$$p(x_1) = \frac{1}{Z} m_j(x_1) \tag{14}$$
   where x_j is the second last in $\mathcal{I}$.

The above algorithm produces one dimensional table that summarize the k different values of $p(x_1)$. This process easily generalize to computation of other marginals. Note that for directed graphs, $Z = 1$. For undirected graphs, if we compute each (unnormalized) value on the LHS of Eq.14, we can sum them to get the value of partition function Z.

The complexity of variable elimination in the above case of computing marginals is $O((n-1)k^r)$, where n is the number of variables, k is the maximum value a variable can take and r is the number of variables participating in largest "intermediate" table. In other words, the overall complexity of variable elimination is determined by the number of the largest elimination clique. However, this number is relevant to elimination ordering $\mathcal{I}$. Tree-width t is one less than the smallest achievable value of the cardinality of the largest elimination clique, ranging over all possible elimination orderings. "good" elimination orderings lead to small cliques and hence reduce the complexity. That finding the best elimination ordering of a graph is NP-hard, which implies that inference is also NP-hard. But there often exist "obvious" optimal or near-optimal elimination ordering.

Before closing this section, it is reasonable to illustrate variable elimination algorithm with a small example and see how it relates to node elimination in the graph:

For all i, suppose that the range of X_i is $\{0, 1, 2\}$. The factor graph is shown in Fig.5

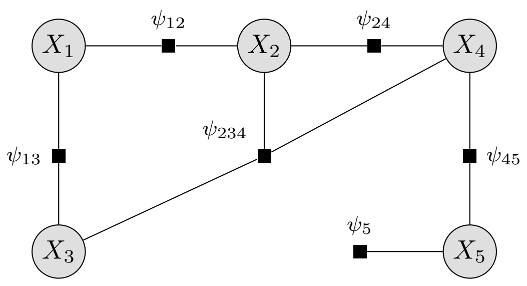
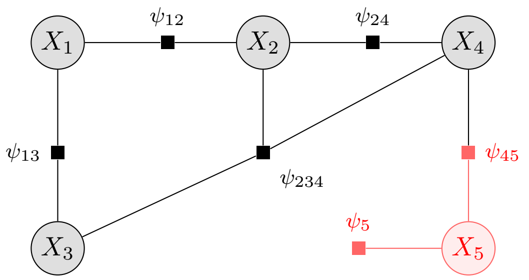


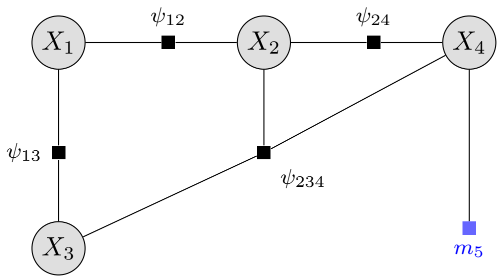
Figure 5: Example factor graph

Let's compute $p(x_1)$ using variable elimination (corresponding node elimination is illustrated in following figures).

$$\begin{aligned}
p(x_1) &= \frac{1}{Z}\sum_{x_2,x_3,x_4,x_5} \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{234}(x_2,x_3,x_4)\psi_{45}(x_4,x_5)\psi_5(x_5) \\
&= \frac{1}{Z}\sum_{x_2,x_3,x_4} \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{234}(x_2,x_3,x_4)\sum_{x_5}\psi_{45}(x_4,x_5)\psi_5(x_5) && \text{Fig.6a} \\
&= \frac{1}{Z}\sum_{x_2,x_3,x_4} \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{234}(x_2,x_3,x_4)m_5(x_4) && \text{Fig.6b} \\
&= \frac{1}{Z}\sum_{x_2,x_3} \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\sum_{x_4}\psi_{24}(x_2,x_4)\psi_{234}(x_2,x_3,x_4)m_5(x_4) && \text{Fig.7a} \\
&= \frac{1}{Z}\sum_{x_2,x_3} \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)m_4(x_2,x_3) && \text{Fig.7b} \\
&= \frac{1}{Z}\sum_{x_2} \psi_{12}(x_1,x_2)\sum_{x_3}\psi_{13}(x_1,x_3)m_4(x_2,x_3) && \text{Fig.8a} \\
&= \frac{1}{Z}\sum_{x_2} \psi_{12}(x_1,x_2)m_3(x_1,x_2) && \text{Fig.8b} \\
&= \frac{1}{Z}\sum_{x_2} \psi_{12}(x_1,x_2)m_3(x_1,x_2) && \text{Fig.9a} \\
&= \frac{1}{Z}m_2(x_1) && \text{Fig.9b}
\end{aligned}$$

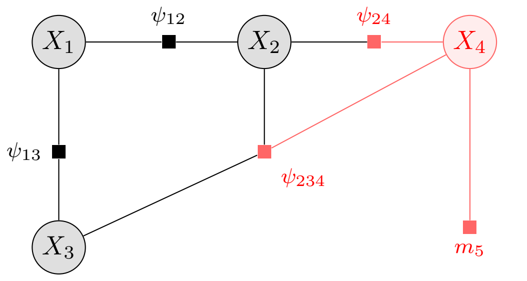


(a) Node elimination step 1.1

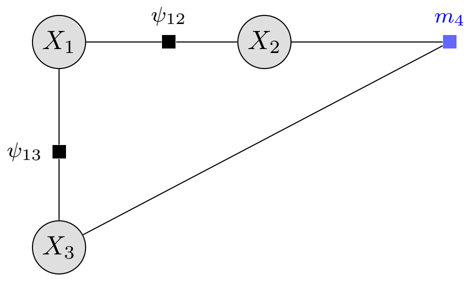


(b) Node elimination step 1.2

Figure 6: Elimination of node X_5

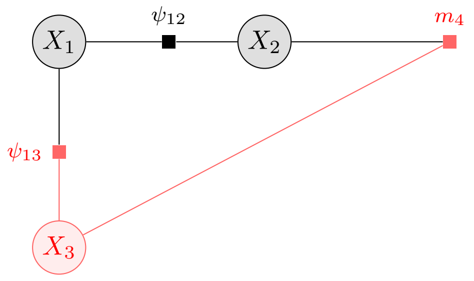


(a) Node elimination step 2.1

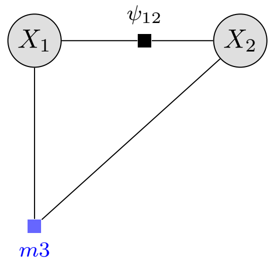


(b) Node elimination step 2.2

Figure 7: Elimination of node X_4



(a) Node elimination step 3.1



(b) Node elimination step 3.2

Figure 8: Elimination of node X_3

2.4 Sum-product belief propagation

Message passing is a great idea in machine learning. For a given node in an acyclic graph, based on counting information passed from all directions, the node is able to compute the global count. Message passing is fundamental in belief propagation.

In a factor graph, neighbors of any variable nodes are factor nodes and neighbors of any factor nodes are variable nodes. If a variable node X_i wants to send a message $\mu_{i\to\alpha}$ to one of its neighbors ψ_α, it has to wait until it receives messages $\mu_{\alpha'\to i}$, $\forall\alpha' \in \mathcal{N}(i)\backslash\alpha$. Then it computes and sends the message:

$$\mu_{i\to\alpha}(x_i) = \prod_{\alpha'\in\mathcal{N}(i)\backslash\alpha} \mu_{\alpha'\to i}(x_i) \tag{15}$$

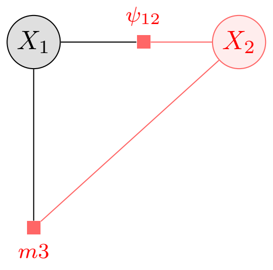
Similarly, if a factor node ψ_α wants to send a message $\mu_{\alpha\to i}$ to one of its neighbors X_i, it has to wait until it receives messages $\mu_{j\to\alpha}$, $\forall j \in \mathcal{N}(\alpha)\backslash i$. Then it computes and sends the message:

$$\mu_{\alpha\to i}(x_i) = \sum_{\mathbf{x}_\alpha:\mathbf{x}_\alpha[i]=x_i} \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\mathcal{N}(\alpha)\backslash i} \mu_{j\to\alpha}(\mathbf{x}_\alpha[i]) \tag{16}$$

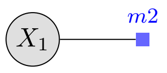
where $\mathbf{x}_\alpha[i]$ is the component corresponding to variable node X_i.

Belief at variable node X_i is

$$b_i(x_i) = \prod_{\alpha\in\mathcal{N}(i)} \mu_{\alpha\to i}(x_i) \tag{17}$$



(a) Node elimination step 4.1



(b) Node elimination step 4.2

Figure 9: Elimination of node X_2

Belief at factor node ψ_α is

$$b_\alpha(\mathbf{x}_\alpha) = \psi_\alpha(\mathbf{x}_\alpha) \prod_{i \in \mathcal{N}(\alpha)} \mu_{i \to \alpha}(\mathbf{x}_\alpha[i]) \tag{18}$$

Fig. 10 illustrates beliefs of and messages sent by variable nodes and factor nodes.

With the above information, we can introduce the sum-product belief propagation algorithm as follows:

Input: An acyclic factor graph
Output: Exact marginals for each variable and factor
Algorithm:

1. Initialize the messages to the uniform distribution $\mu_{i \to \alpha}(x_i) = 1, \quad \mu_{\alpha \to i}(x_i) = 1$
2. Choose a root node
3. Send messages from the *leaves* to *root* using Eq.15,16
   Send messages from the *root* to *leaves* using Eq.15,16
4. Compute the beliefs (unnormalized marginals) using Eq.17,18
5. Normalize beliefs and return the exact marginals: $p_i(x_i) \propto b_i(x_i)$, $p_\alpha(\mathbf{x}_\alpha) \propto b_\alpha(\mathbf{x}_\alpha)$

Note that a node computes an outgoing message along an edge only after it has received incoming messages along all its other edges. (Acyclic) Belief propagation can be viewed as dynamic programming in the following way: If you want the marginal $p_i(x_i)$ where X_i is a variable node of degree k, you can think of that summation as a product of k marginals computed on smaller subgraphs. Each subgraph is obtained by cutting some edge of the tree. The message passing algorithm uses dynamic programming to compute the marginals on all such subgraphs, working from smaller to larger. So you can compute all the marginals.

2.5 Max-product belief propagation

Sum-product belief propagation can be used to compute the marginals: $p_i(x_i)$. Max-product belief propagation is a variant of Sum-product belief propagation that can be used to compute the most likely assignment: $\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} p(\mathbf{x})$. If we replace the summation in Eq.15 with max function and hold everything else unchanged in sum-product belief propagation, we can get max-product belief propagation with a new message

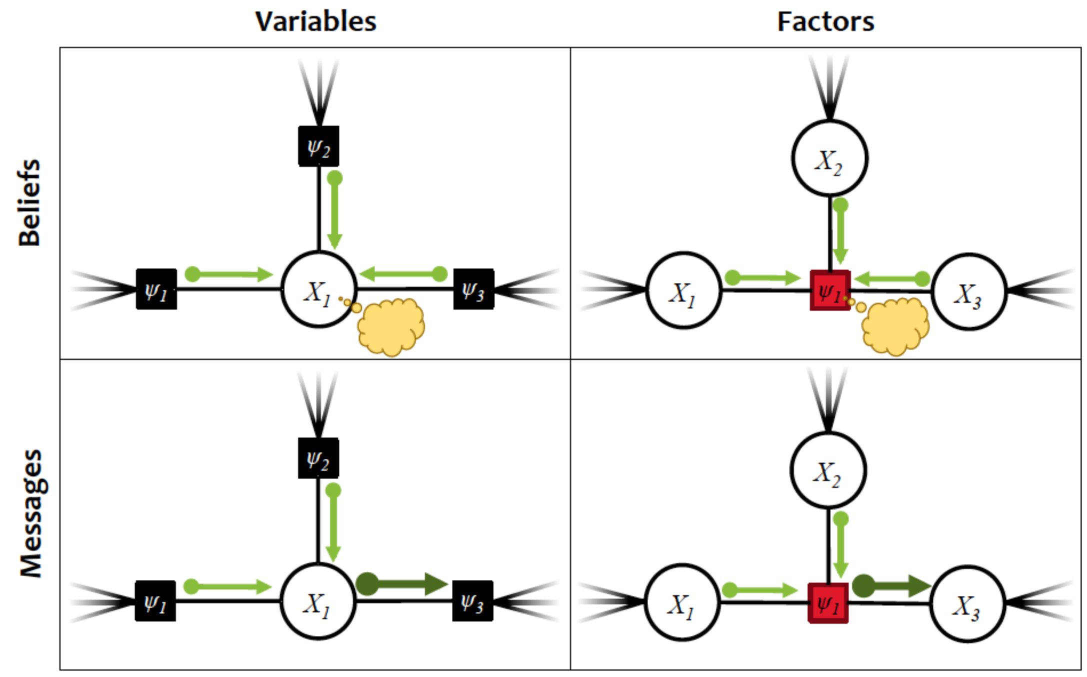


Figure 10: Beliefs and messages of variables and factors.

passed from factor nodes to variable nodes:

$$\mu_{\alpha\to i}(x_i) = \max_{\mathbf{x}_\alpha : \mathbf{x}_\alpha[i]=x_i} \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\mathcal{N}(\alpha)\setminus i} \mu_{j\to\alpha}(\mathbf{x}_\alpha[i]) \tag{19}$$

The max-marginal $b_i(x_i)$ from max-product belief propagation is the (unnormalized) probability of the MAP assignment under the constraint $X_i = x_i$. Assuming no ties, the MAP assignment for an acyclic graph is given by

$$\hat{x}_i = \arg\max_{x_i} b_i(x_i) \tag{20}$$

In order for smooth transition from sum-product to max-product, we introduce *deterministic annealing*:

1. Incorporate inverse temperature parameter into each factor, from which we get the annealed joint distribution:
$$p(\mathbf{x}) = \frac{1}{Z}\prod_\alpha \psi_\alpha(\mathbf{x}_\alpha)^{\frac{1}{T}} \tag{21}$$

2. Send messages as usual for sum-product belief propagation

3. Anneal T from 1 to 0:

$T = 1$	Sum-product
$T \to 0$	Max-product

4. Take resulting beliefs to power T

The reason that the algorithm works 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: $a \cdot b + a \cdot c = a \cdot (b + c)$. This distributive law plays a key role in sum-product belief propagation, in which "sum" operator repeatedly migrates across the "product" operator. The ability to group and reorder intermediate factors is also required. It can be shown that associativity, commutativity and distributivity are all that are needed for sum-product algorithm. We can also check that "product" distributes over "max": $\max(a \cdot b, a \cdot c) = a \cdot max(b, c)$.

A practical problem is that multiplying many small numbers together can underflow. Thus, instead of using sum-product, we use log-add-sum; and instead of using max-product, we use max-sum.