CS7140: Advanced Machine Learning **Spring 2020**

Lecture 3: Exact Inference and Message Passing

Lecturer: Rose Yu *Scribes: Jing Xie, Emmanuel Ojuba*

3.1 Message Passing

3.1.1 Counting Soldiers

The commander wishes to count the number of soldiers in a straight line. There are two ways the problem can be solved:

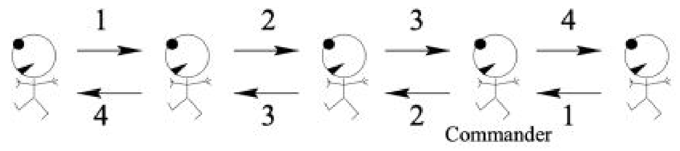

Figure 3.1: Counting Soldiers

1. The commander counts all the soldiers himself. This method is inefficient and requires global communication channels between the commander and all the soldiers
2. Message passing scheme as follows[KF09]:
   (a) If you are the front soldier, say ”one” to the soldier behind you
   (b) If you are the rearmost soldier, say ”one” to the soldier in front of you
   (c) If a soldier ahead of or behind you says a number to you, add one to it, and say the new number to the soldier on the other side.

   The second solution only requires local communication hardware and simple computations

3.1.2 Separation

3.1.2.1 Cycles

If the commander wishes to count the number of soldiers in a network with at least one cycle, the message passing algorithm we introduced in the previous section does not work. The message passing algorithm uses the property that the total number of soldiers can be divided into the sum of number of soldiers behind a point and the number of soldiers ahead. In this case, it is not obvious who is ’ahead’ and who is ’behind’.

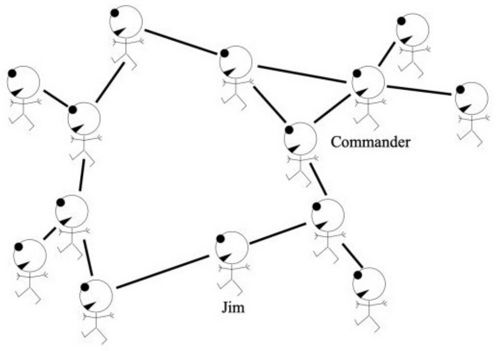

Figure 3.2: Cycles

3.1.2.2 Trees

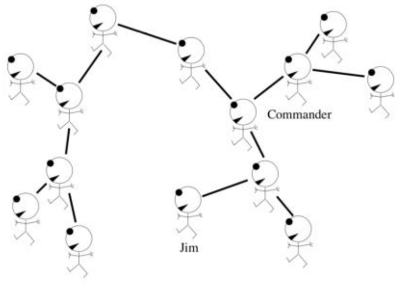

Figure 3.3: Tree Message Passing

In trees, a message passing algorithm can be used to count the number of soldiers. The message passing algorithm is as follows [KF09]:

1. Count your number of neighbors, N
2. Count the number of messages you have received from your neighbors, m, and the values $v_1, v_2, ..., v_N$. V is the running total of messages you have received
3. If the number of messages received, m, equals $N-1$, identify the neighbor who has not sent a message and tell them $V+1$
4. If the number of messages you have received, m, equals N, then:
   (a) the number $V+1$ is the required total.
   (b) say to each neighbor n the number $V+1-v_n$

3.1.2.3 Path Counting

A message passing algorithm can be used to answer the following questions about the grid in Fig 3.4:

- How many paths are there from A to B?
- If a random path from A to B is selected, what is the probability that it passes through a particular

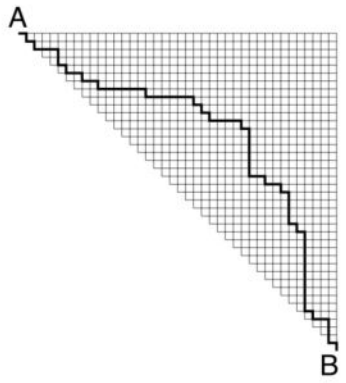

Figure 3.4: Path Counting

node in the grid?

- How can a random path from A to B be selected?

The Number of Paths from A to B

In the general scenario, it is inefficient to enumerate all the number of possible paths from A to B. The computational load is reduce when we realized that every path from A to a selected point P must come through one or both of its upstream neighbors (neighbors above and left). The number of paths from A to P can be found by adding the number of paths to its left and above neighbors [KF09].

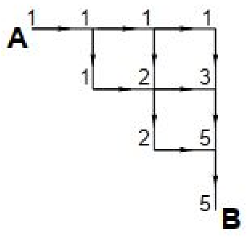

Figure 3.5: Path Counting

The message passing algorithm is as follows [KF09]:

1. We send '1' message from A
2. When a node has received messages from all its upstream neighbors, it sends the sum to its downstream neighbors.
3. At B the number of paths emerges

The Probability of Passing through a Node

The probability of passing through node, P is the number of nodes that pass through P divided by the total number of nodes.

We evaluate the number of paths that pass through node P as the number of paths from A to P in the *forward pass* multiplied by the number of nodes from B to P in the backward pass.

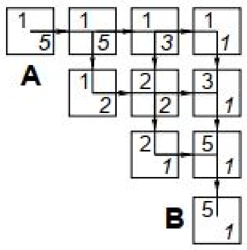

Figure 3.6: Number of paths through P. P is the center node, so the number of paths = 2 x 2 = 4

3.1.2.4 The Lowest Cost Path Problem

We utilize a message passing algorithm, known as **min-sum** or **Viterbi Algorithm** As in the case of the number of paths problem, we avoid enumerating all the possible paths from A to B. We utilize a message passing find the lowest cost path from A to intermediate points between A and B. We repeat this procedure until we arrive at B [KF09].

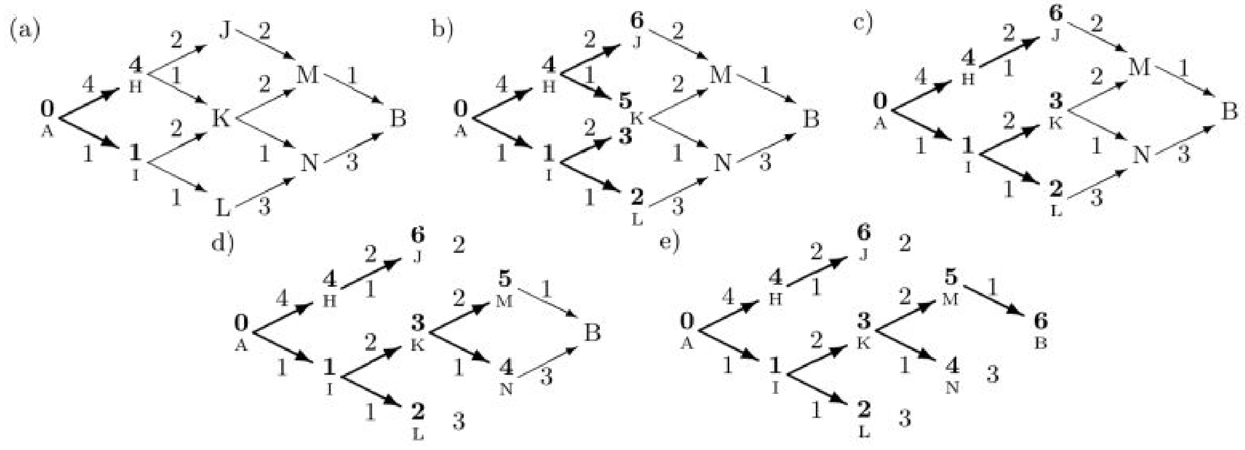

Figure 3.7: Lowest Cost Path

3.2 Exact Marginalization in Graphs

3.2.1 The General Problem

Assume that a function P^* of a set of N variables $\mathbf{x} \equiv \{x_n\}_{n=1}^N$ is defined as a product of M factors as follows

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{3.1}$$

Each of the factors $f_m(\mathbf{x}_m)$ is a function of a subset $\mathbf{x}_m$ of the variables that make up $\mathbf{x}$. If P^* is a positive function then we may be interested in a normalized function [KF09]:

$$Z = \sum_{\mathbf{x}} \equiv \frac{1}{Z} P^*(\mathbf{x}) = \frac{1}{Z} \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{3.2}$$

where the normalizing constant Z is defined by

$$Z = \sum_{x} \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{3.3}$$

3.2.1.1 The Normalization Problem

The first task to be solved is to compute the normalizing constant Z.

$$Z = \sum_{\mathbf{x}} \equiv \frac{1}{Z} P^*(\mathbf{x}) = \frac{1}{Z} \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{3.4}$$

3.2.1.2 The Marginalization Problem

The second task to be solved is to compute the marginal function of any variable x_n defined by

$$Z_n(x_n) = \sum_{\{x_{n'}\}, n' \neq n} P^*(\mathbf{x}) \tag{3.5}$$

3.2.2 Variable Elimination

Let us analyze the complexity of the process on a general chain shown in figure (3.8). Assume that we have a chain with n variables $X_1 \rightarrow ... \rightarrow X_T$, where each variable in the chain has k values. As described, the algorithm would compute $P(X_{i+1})$ from $P(X_i)$, for i = 1,...,T-1. Each such step would consist of the following computation:

$$P(X_{i+1}) = \sum_{x_i} P(X_{i+1}|x_i)P(x_i) \tag{3.6}$$

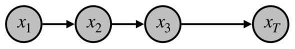

Figure 3.8: Example of a Network

where $P(X_i)$ is computed in the previous step. The cost of each such step is $O(k)$. The total cost is $O(nk)$. By comparison, consider the process of generating the entire joint and summing it out, which requires that we generate k^T probabilities for the different events x_i,...,x_n.

The two ideas that help us address the exponential blowup of the joint distribution are:

- Because of the structure of the Bayesian network, some subexpressions in the joint depend only on a small number of variables.

- By computing these expressions once and caching the results, we can avoid generating them exponentially many times.

The general formula used in variable elimination is as follows:

$$\begin{aligned} p(x_T) &= \sum_{x_2,\ldots,x_{T-1}} p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1}) \\ &= \sum_{x_{T-1}} p(x_T|x_{T-1}) \sum_{x_{T-2}} p(x_{T-1}|x_{T-2}) \ldots \sum_{x_1} p(x_2|x_1)p(x_1) \\ &= \sum_{x_{T-1}} p(x_T|x_{T-1}) \sum_{x_{T-2}} p(x_{T-1}|x_{T-2}) \ldots \sum_{x_2} p(x_3|x_2)\tau_2 \end{aligned} \tag{3.7}$$

where we define $\tau_t : \; Val(x_{t+1}) \mapsto \mathbb{R}$

Example: We begin by considering the inference task in a very simple network $A \to B \to C \to D$. To compute P(D), we need to sum together all of the entries where $D = d^1$, and to (separately) sum together all of the entries where $D = d^2$, and perform the computation for all values d^n that D could take. For an illustration, we show in figure (3.9), the exact computation that needs to be performed, for binary-valued variables A,B,C,D. Examining this summation, we see that it has a lot of structures. For example, the third and fourth terms in the first two entries are both $P(c^1|b^1)P(d^1|c^1)$ in figure (3.9). We can therefore modify the computation to first compute $P(a^1)P(b^1|a^1) + P(a^2)P(b^1|a^2)$ and only then multiply by the common term. The same structure is repeated throughout the table. If we perform the same transformation, we get a new expression, as shown in figure (3.10).

We now observed that certain terms are repeated several times in this expression. Specifically $P(a^1)P(b^1|a^1)+ P(a^2)P(b^1|a^2)$ are each repeated four times. Thus, we can set $\tau_1(b^1) = P(a^1)P(b^1|a^1) + P(a^2)P(b^1|a^2)$ in figure (3.11).

$$\begin{array}{rllll}
 & P(a^1) & P(b^1 \mid a^1) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & P(a^1) & P(b^1 \mid a^1) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
\\
 & P(a^1) & P(b^1 \mid a^1) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & P(a^1) & P(b^1 \mid a^1) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & P(a^2) & P(b^1 \mid a^2) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & P(a^1) & P(b^2 \mid a^1) & P(c^2 \mid b^2) & P(d^2 \mid c^2) \\
+ & P(a^2) & P(b^2 \mid a^2) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}$$

Figure 3.9: Computing P(D) by summing over the joint distribution in the chain

$$\begin{array}{rlll}
 & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
\\
 & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & (P(a^1)P(b^1 \mid a^1) + P(a^2)P(b^1 \mid a^2)) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & (P(a^1)P(b^2 \mid a^1) + P(a^2)P(b^2 \mid a^2)) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}$$

Figure 3.10: The First transformation. Here we factorize the terms by pulling out the expressions $P(a^1)P(b^1|a^1) + P(a^2)P(b^1|a^2)$ and $P(a^1)P(b^2|a^1) + P(a^2)P(b^2|a^2)$

$$\begin{array}{rlll}
 & \tau_1(b^1) & P(c^1 \mid b^1) & P(d^1 \mid c^1) \\
+ & \tau_1(b^2) & P(c^1 \mid b^2) & P(d^1 \mid c^1) \\
+ & \tau_1(b^1) & P(c^2 \mid b^1) & P(d^1 \mid c^2) \\
+ & \tau_1(b^2) & P(c^2 \mid b^2) & P(d^1 \mid c^2) \\
\\
 & \tau_1(b^1) & P(c^1 \mid b^1) & P(d^2 \mid c^1) \\
+ & \tau_1(b^2) & P(c^1 \mid b^2) & P(d^2 \mid c^1) \\
+ & \tau_1(b^1) & P(c^2 \mid b^1) & P(d^2 \mid c^2) \\
+ & \tau_1(b^2) & P(c^2 \mid b^2) & P(d^2 \mid c^2)
\end{array}$$

Figure 3.11: The Second Transformation. Here we set $\tau_1(b^1) = P(a^1)P(b^1|a^1) + P(a^2)P(b^1|a^2)$

$$\begin{array}{rll}
 & (\tau_1(b^1)P(c^1 \mid b^1) + \tau_1(b^2)P(c^1 \mid b^2)) & P(d^1 \mid c^1) \\
+ & (\tau_1(b^1)P(c^2 \mid b^1) + \tau_1(b^2)P(c^2 \mid b^2)) & P(d^1 \mid c^2) \\
\\
 & (\tau_1(b^1)P(c^1 \mid b^1) + \tau_1(b^2)P(c^1 \mid b^2)) & P(d^2 \mid c^1) \\
+ & (\tau_1(b^1)P(c^2 \mid b^1) + \tau_1(b^2)P(c^2 \mid b^2)) & P(d^2 \mid c^2)
\end{array}$$

Figure 3.12: The Third Transformation. We recursively repeat the process of fig 3.10 for variable c

$$\begin{array}{rll}
 & \tau_2(c^1) & P(d^1 \mid c^1) \\
+ & \tau_2(c^2) & P(d^1 \mid c^2) \\
\\
 & \tau_2(c^1) & P(d^2 \mid c^1) \\
+ & \tau_2(c^2) & P(d^2 \mid c^2)
\end{array}$$

Figure 3.13: The Fourth Transformation. We recursively repeat the process of fig 3.11 for variable c

To summarize:

- Push in certain variables into the product
- Eliminate a variable at each time
- Different orderings may dramatically alter the running time
- It is NP-hard to find the best ordering

3.2.3 Sum-product Algorithm

3.2.3.1 Factor Graphs

A factor graph F is an undirected graph containing two types of nodes: variable nodes and factor nodes. Each factor $f_m(\mathbf{x}_m)$ is a function of a subset $\mathbf{x}_m$ of the variables that constitute $\mathbf{x}$. The graph only contains edges between variable nodes and factor nodes. A factor graph F is parameterized by a set of factors, where each factor node f_m is associated with precisely one factor m, whose scope is the set of variables that are neighbours of f_m in the graph. An example of a factor graph is shown in the figure (3.14).

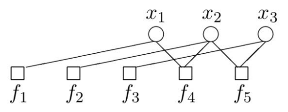

Figure 3.14: Factor graph

3.2.3.2 Rules in Sum-product Algorithm

The sum-product algorithm will involve messages of two types passing along the edges in the factor graph:

- messages $q_{n\to m}$ from variable nodes to factor nodes.
- messages $r_{m\to n}$ from factor nodes to variable nodes.

A message (or either type, q or r) that is sent along an edge connecting factor f_m to variable x_n is always a function of the variable x_n. The rules for updating the two sets of messages as as follows:

From variable to factor:

$$q_n(x_n) = \prod_{m' \subset M(n) \setminus m} r_{m' \to n}(x_n) \tag{3.8}$$

From factor to variable:

$$r_{m\to n} = \sum_{x_m \setminus n} (f_m(\mathbf{x}_m) \prod_{n' \subset N(m) \setminus n} q_{n' \to m}(x_{n'})) \tag{3.9}$$

where we identify the set of variables that the mth factor depends on, x_m, by the set of their indices $N(m)$. We define the set of factors in which variable n participates, by $M(n)$. We denote a set $N(m)$ with variable n excluded by $N(m) \setminus n$. We introduce the shorthand $\mathbf{x}_m \setminus n$ to denote the set of variables in $\mathbf{x}_m$ with x_n excluded.

3.2.3.3 Starting and Finishing the Sum-Product Algorithm, method 1

If the graph is tree-like then it must have nodes that are leaves. These leaf nodes can broadcast their messages to their respective neighbours from the start [KF09].

For all leaf variable nodes n:

$$q_{n\to m}(x_{n'}) = 1 \tag{3.10}$$

For all leaf variable nodes m:

$$r_{m\to m}(x_n) = f_m(x_n) \tag{3.11}$$

We can then adopt the procedure: a message is created in accordance with the rules(3.8,3.9) only if all the messages on which it depends are present. Messages will thus flow through the tree, one in each direction along every edge, and after a number of steps equal to the diameter of the graph, every message will have been created. The answers we require can then be read out [KF09].

The **marginal function of $\boldsymbol{x_n}$** is obtained by multiplying all the incoming messages at that node:

$$Z_n(x_n) = \prod_{m\subset M(n)} r_{m\to n}(x_n) \tag{3.12}$$

The **normalizing constant $\boldsymbol{Z}$** can be obtained by summing any marginal function:

$$Z = \sum_{x_n} Z_n(x_n) \tag{3.13}$$

The **normalized marginals** obtained from:

$$P_n(x_n) = \frac{Z_n(x_n)}{Z} \tag{3.14}$$

3.2.3.4 Starting and Finishing the Sum-Product Algorithm, method 2

The algorithm can be initialized by setting all the initial messages from variables to 1:

for all n,m:

$$q_{n\to m}(x_n) = 1 \tag{3.15}$$

then proceeding with the factor message update rule (3.9), alternating with the variable message update rule (3.8). The reason for introducing this lazy method is that it can be applied to graphs that are not tree-like [KF09].

3.2.3.5 A Factorization View of the Sum-product Algorithm

One way to view the sum-product algorithm is that it re-expresses the original factored function, the product of M factors [KF09] $P^*(x) = \prod_{m=1}^{M} f_m(\mathbf{x}_m)$, as another factored function which is the product of $M+N$ factors,

$$P^*(x) = \prod_{m=1}^{M} \phi_m(\mathbf{x}_m) \prod_{n=1}^{N} \psi_n(x_n) \tag{3.16}$$

Each factor ϕ_m is associated with a factor node m, and each factor $\psi_n(x_n)$ is associated with a variable node. Initially $\phi_m(\mathbf{x}_m) = f_m(\mathbf{x}_m)$ and $\psi_n(x_n) = 1$.

Each time a factor-to-variable message $r_{m\to n}(x_n)$ is sent, the factorization is updated thus:

$$\psi_n(x_n) = \prod_{m\subset M(n)} r_{m\to n}(x_n) \tag{3.17}$$

$$\phi_m(\mathbf{x}_m) = \frac{f(\mathbf{x}_m)}{\prod_{n \subset N(m)} r_{m\to n}(x_n)} \tag{3.18}$$

And each message can be computed in terms of ϕ and ψ using

$$r_{m\to n}(x_n) = \sum_{\mathbf{x}_m \backslash n} (\phi_m(\mathbf{x}_m) \prod_{n' \subset N(m)} \psi_{n'}(x_{n'})) \tag{3.19}$$

which differs from the assignment (3.9) in that the product is over all $n' \subset N(m)$.

This factorization viewpoint applies whether or not the graph is tree-like.

3.2.4 The min-sum algorithm

The Maximization Problem: Find the setting of $\mathbf{x}$ that maximizes the product $P^*(\mathbf{x})$.

Replace Operations: This problem can be solved by replacing the two operations **add** and **multiply** everywhere they appear in the sum-product algorithm by another pair if operations that satisfy the distributive law, namely **max** and **multiply**. Thus, the sum-product algorithm can be turned into a max-product algorithm that computes $\max_{\mathbf{x}} \mathrm{P}^*(\mathbf{x})$ [KF09].

In practice, the max-product algorithm is most often carried out in the **negative log likelihood domain**, where **max** and **product** become **min** and **sum**.

The min-sum algorithm is also known as the Viterbi algorithm.

3.2.5 The Junction Tree Algorithm

When the factor graph is not a tree, there are several exact and approximate methods for marginalization on graphs.
The most widely used exact method for handling marginalization on graphs with cycles is called the **junction tree algorithm**. This algorithm works by agglomerating variables together until the agglomerated graph has no cycles.

Here are the four steps that we use independence to convert graphs to trees.

1. **Moralize the graph:** directed to undirected
2. **Triangulate the graph:** separation
3. **Build a junction tree:** graph to tree
4. **Message passing:** sum product algorithm

3.2.5.1 Moralization

Convert a directed graph to a undirected graph:

- Connect nodes that have common children

- Drop the arrows

Figure (3.15) shows an example of moralization.

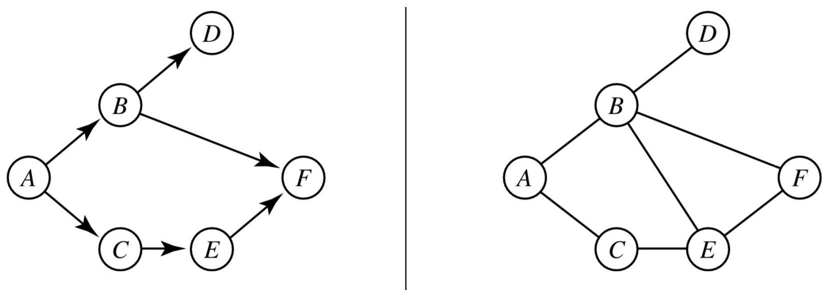

Figure 3.15: An Example of Moralization

3.2.5.2 Triangulation

Make the graph chordal(no cycle/loops)

- Add links until there is no chordless cycle of 4 or more nodes
- Every induced cycle has exactly 3 nodes

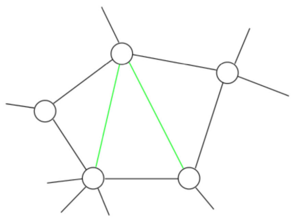

Figure 3.16: An Example of Triangulation

3.2.5.3 Build a Junction Tree

- Each node is a clique of variables, each edge is a potential function
- Separator nodes contain clique intersection of variables

3.2.5.4 Message Passing

- Write a junction tree as potentials of its nodes like figure (3.17)
- Variable Elimination
- Message passing: Use sum-product algorithm rule(3.8, 3.9)

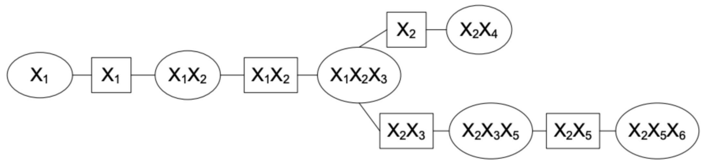

Figure 3.17: Building a Junction Tree

References

[KF09] D. MacKay, "Information Theory, Inference, and Learning Algorithms " *Cambridge University Press*, 1987, pp. 241–247, 334–340.