Inference in Probabilistic Graphical Models

Prof. Nicholas Zabaras
School of Engineering
University of Warwick
Coventry CV4 7AL
United Kingdom

Email: nzabaras@gmail.com
URL: http://www.zabaras.com/

August 28, 2014
Contents

- Introduction, Inference on a Chain, Message Passing: Chapman-Kolmogorov Equations, Computing All Marginals, Max Product Algorithm on a Chain
- Factor Graphs, Sum Product Algorithm for Factor Trees
- Sum-Product Like Algorithm for Polytrees, Belief Propagation
- Max-Sum Algorithm, Max Vs. ArgMax, Map Eliminate, Max-Product, Max-Sum, Back Track, Inference in Multiple Connected DAGs
- Inference in General Graphs: The Junction Tree Algorithm, Loopy Belief Propagation, Learning the Graph Structure

- Kevin Murphy, *Machine Learning: A probabilistic Perspective*, Chapter 19
- Chris Bishop, *Pattern Recognition and Machine Learning*, Chapter 8
- Video Lectures on Machine Learning, Z. Gahramani, C. Bishop and others.
The Elimination Algorithm allow us to compute a single marginal.

The Sum-Product algorithm (also known as Belief Propagation) will allow us to compute all marginals and conditionals. They key idea of the algorithm is the development of a calculus of intermediate factors common to many elimination orderings. We will also discuss computing maximum a posteriori probabilities.

However, the Sum-Product algorithm works only for trees and tree-like graphs (using the factor approach we can handle polytrees - a class of directed graphical models where nodes have multiple parents).

In another lecture, we will generalize to the junction tree algorithm (exact inference problem).

Before starting our discussion with trees, we first introduce ideas through a one-dimensional chain like graph.
Inference on a Chain

- Consider an undirected chain (many missing links, tree structure)

\[ 
\begin{array}{cccc}
  x_1 & x_2 & \ldots & x_{N-1} & x_N \\
\end{array} 
\]

- The marginal distribution is given as

\[
p(x_n) = \sum_{x_1} \cdots \sum_{x_{n-1}} \sum_{x_{n+1}} \cdots \sum_{x_N} p(x)
\]

with a cost of \( O(M^N) \) if we have \( M \) configurations per variable (exponential in the length of the chain)

- But we can exploit the structure of the graph since the joint distribution is given as:

\[
p(x) = \frac{1}{Z} \psi_{1,2}(x_1, x_2) \psi_{2,3}(x_2, x_3) \cdots \psi_{N-1,N}(x_{N-1}, x_N)
\]
Inference on a Chain: Message Passing

The $\mu_\alpha$ term is computed by working forward across the chain. The $\mu_\beta$ term works backwards. The cost is now $O(NM^2)$ (N Tables of 2 variables) - linear in $N$. Can be extended to any graph with no loops.
Chapman-Kolmogorov Equations

- $\mu_\alpha(x_{n-1})$ is a message passed forwards along the chain from node $x_{n-1}$ to node $x_n$.
- Similarly, $\mu_\beta(x_n)$ can be viewed as a message passed backwards along the chain to node $x_n$ from node $x_{n+1}$.
- These messages comprise each a set of $M$ values.
- $\mu_\alpha(x_n)$ and $\mu_\beta(x_n)$ can be evaluated recursively.

\begin{align*}
\mu_\alpha(x_n) &= \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \left[ \sum_{x_{n-2}} \ldots \right] = \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \mu_\alpha(x_{n-1}) \\
\text{Initialize: } \mu_\alpha(x_2) &= \sum_{x_1} \psi_{1,2}(x_1, x_2) \\
\mu_\beta(x_n) &= \sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1}) \left[ \sum_{x_{n+2}} \ldots \right] = \sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1}) \mu_\beta(x_{n+1})
\end{align*}

Algorithm: Computing All Marginals

- To compute all local marginals (cost of $O(2 \times NM^2)$):
  - Compute and store all forward messages, $\mu_\alpha(x_n)$.
  - Compute and store all backward messages, $\mu_\beta(x_n)$.
  - Compute $Z$ once at any node $x_n$ ($\sum_{x_n} \mu_\alpha(x_n) \mu_\beta(x_n)$, cost $O(M)$)
  - Compute $p(x_n) = \frac{1}{Z} \mu_\alpha(x_n) \mu_\beta(x_n)$ for all variables required.
  - Using these marginals, one can find the $x_n$ that maximizes $p(x_n)$

- The joint distribution for two neighbouring nodes $x_{n-1}$ and $x_n$ on the chain can also be computed as

$$p(x_{n-1}, x_n) = \frac{1}{Z} \mu_\alpha(x_{n-1}) \psi_{n-1,n}(x_{n-1}, x_n) \mu_\beta(x_n)$$
Algorithm: Computing Conditionals

- Assume we want to compute $p(x_n|x_N)$ for all $n=1,...,N-1$.
  - One needs to only change the initial message for the $\beta$-recursion:
    \[
    \mu_\beta(x_{N-1}) = \psi_{N-1,N}(x_{N-1}, \bar{x}_N)
    \]
  - Note that there is no summation now on $x_N$.
  - The $\alpha$- and $\beta$- recursions remain the same apart from the above change and one computes for each $n=1,...,N-1$:
    \[
    p(x_n | \bar{x}_N) = \frac{1}{Z} \mu_\alpha(x_n) \mu_\beta(x_n)
    \]
Our goal here is to find the mode of the joint distribution:

\[ p(x_{\text{max}}) = \max_x p(x) = \max_{x_1} \ldots \max_{x_N} p(x_1, \ldots, x_N) \]

\[ = \frac{1}{Z} \max_{x_1} \ldots \max_{x_N} \left[ \psi_{1,2}(x_1, x_2) \ldots \psi_{N-1,N}(x_{N-1}, x_N) \right] \]

\[ = \frac{1}{Z} \max_{x_1} \left[ \max_{x_2} \left[ \psi_{1,2}(x_1, x_2) \left[ \ldots \max_{x_N} \psi_{N-1,N}(x_{N-1}, x_N) \right] \right] \right] \]

- Message passing algorithm (linear in the number of nodes) with `sum' replaced by `max'
- This is a generalization of the Viterbi algorithm for HMMs
- Define

\[ \phi(x_i) = \max_{x_1} \ldots \max_{x_{i-1}} \max_{x_{i+1}} \ldots \max_{x_N} p(x_1, x_2, \ldots, x_N) \]

- Then

\[ x_i^{\text{MAP}} = \arg \max_{x_i} \phi(x_i) \]

- We will discuss this further later in this lecture.
Sum-Product Algorithm

- The elimination algorithm discussed in an earlier lecture works for all graphs, but it must be run separately for each single-node marginal.

- The sum-product algorithm efficiently computes all marginals in the special case of trees (already shown earlier for the case of one-dimensional chains).

- Using factor graphs, we can extend the sum-product algorithm to polytrees.

- One can generalize for graphs using the so called junction tree algorithm.
Trees

- **Undirected Tree**: There is only one path between any pairs of nodes.

- **Directed Tree**: Its moralized graph is an undirected tree.

- **Directed and undirected trees make the same CI assumptions**

- This is why moralization of a directed tree adds no edges

- However, **undirected trees are not locally normalized**

- The two are essentially equivalent for the purposes of probabilistic inference

- We focus on the undirected case
Inference on Trees

- **A tree** is an undirected graph in which there is only and only one path between any pair of nodes.

- **A directed graph** is a tree when its moralized graph is a tree. Directed trees have a single node that has no parent (the root node) and that all other nodes have exactly one parent.

- **A polytree** is not a directed tree. It has nodes with multiple parents and its moralized graph has loops.

- Any undirected tree can be converted to a directed tree - choose a root node and orient all edges pointing away from the root. The two have the same conditional independence relations.
Parametrization

- For undirected graphs, the cliques are pairs of nodes and single nodes.

\[ p(x) = \frac{1}{Z} \prod_{i \in \mathcal{V}} \psi(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j), \text{ for a tree } \mathcal{T}(\mathcal{V}, \mathcal{E}) \]

- For directed trees:

\[ p(x) = p(x_r) \prod_{(i,j) \in \mathcal{E}} p(x_j | x_i), \text{ where } (i, j) = \text{directed edge}, \{i\} = \pi_j \]

- We can represent the directed tree as undirected tree:

\[ \psi(x_r) = p(x_r), \psi(x_i) = 1 \quad \forall i \neq r, Z = 1 \]
\[ \psi(x_i, x_j) = p(x_j | x_i), \{i\} = \pi_j \]
Recall the definition of evidence potentials

\[ \psi^E_i(x_i) \triangleq \begin{cases} \psi_i(x_i) \delta(x_i, \bar{x}_i), & i \in E \\ \psi_i(x_i), & i \notin E \end{cases} \]

Using this, we can write conditional distributions as follows:

\[
p(x|\bar{x}_E) = \frac{1}{Z^E} \prod_{i \in \mathcal{V}} \psi^E_i(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j), \text{ for a tree } \mathcal{T}(\mathcal{V}, \mathcal{E})
\]

where the normalization factor is

\[
Z^E = \sum_x \left( \prod_{i \in \mathcal{V}} \psi^E_i(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j) \right)
\]

Note that with this parametrization, the conditional and unconditional cases are practically the same (but note $Z^E$ is not 1 in the conditional case).
We specialize the elimination algorithm to trees.

- Treat $f$ as the root and view the tree as a directed tree by directing all edges of the tree to point away from $f$.
- Consider an elimination ordering in which a node is eliminated only after all of its children in the directed version of the tree are eliminated.
- This elimination proceeds inwards from the leaves and generates elimination cliques of size at most two (tree-width=1).
Pruning Leaves

- In the directed case, any unobserved leaf (or subtree) can be pruned from the tree.
  
  - The reason is that the corresponding summation must reduce to unity.
    \[ \sum_{x_j} p(x_j \mid x_i) = 1 \]

- In the unconditional case, we can arrange things so all sums are of this form (elimination ordering that begins with the leaves and proceeds to the root) and \( Z = 1 \) (Something similar happens with undirected graphs even though \( Z \) is affected).

- When we condition, however, the resulting product of potentials is unnormalized (and \( Z^E \) not equal to 1). This brings the directed case closer to the undirected case.

- We can prune any subtree that contains only variables that are not conditioned on by eliminating backwards.
Pruning Leaves

- Thus, we can assume that all leaves are observed (evidence nodes) and all of the sums have to computed explicitly.

- There is no difference in this case between directed and undirected cases.

- It may also be reasonable to assume all nonleaves are unobserved, because observed nonleaves would split the tree; thus $E$ would equal the set of leaves.

- Conditioning on a set of nodes creates conditional independencies through graph separation that will subdivide the graph into smaller trees whose marginal probabilities are solved independently of each other.

- To work with only one tree, we treat the evidence by modifying the definition of the self potentials.
Message Passing

- Consider the case of computing the marginal probability of an arbitrary ancestral node in an undirected tree by elimination.

- An optimal elimination ordering proceeds from leaves to root (postorder) in a tree rooted at the query node.

- Under such an ordering, the intermediate functions generated during the elimination process will pass like messages from descendants to ancestors.

- Each message summarizes evidence in the subtree beneath the node.

- Messages can be reused from one node to the next.
Recall the general elimination algorithm

- Choose an elimination $I$ in which the query node $f$ is the final node
- Place all potentials on an active list
- 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 on the active list.

Consider *depth-first traversal of the tree*. A node is eliminated only after all of its children in the directed version of the tree are eliminated. This proceeds inwards from the leaves and generates cliques of size 2 (tree-width=1). Let us see an example.
A Message Passing Example

\[
p(x_3 | x_2, x_4, x_5) = \frac{1}{Z_E} \prod_{i \in \psi} \psi^E(x_i) \prod_{(i,j) \in \mathcal{E}} \psi(x_i, x_j)
\]

\[
= \frac{1}{Z_E} \sum_{x_1} \psi^E x_3 \psi^E x_1 \psi(x_1, x_3) \psi^E x_2 \psi(x_2, x_1) \psi^E x_4 \psi(x_4, x_3) \psi^E x_5 \psi(x_5, x_3)
\]

\[
= \frac{1}{Z_E} \psi^E x_3 \left( \sum_{x_1} \psi^E x_1 \psi(x_1, x_3) m_{21} x_1 \right) m_{43} x_3 m_{53} x_3
\]

\[
= \frac{1}{Z_E} \psi^E x_3 m_{13} x_3 m_{43} x_3 m_{53} x_3
\]

\[
= \frac{\psi^E x_3 m_{13} x_3 m_{43} x_3 m_{53} x_3}{\sum_{x_3} \psi^E x_3 m_{13} x_3 m_{43} x_3 m_{53} x_3}
\]
Consider the node $i$ closer to the root than node $j$.

What is the intermediate factor when $j$ is eliminated?

$m_{ji}(x_i) = \text{message that } j \text{ sends to } i$

$$
\sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}(x_j) \right)
$$

Inference results in a coupled set of eqs for $m_{ji}(x_i)$ (Complexity $O(m^2)$, $m$=states per node)

For the final node $f$, we have:

$$
p(x_f | x_E) = \frac{\psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}(x_f)}{\sum_{x_f} \psi^E(x_f') \prod_{e \in \mathcal{N}(f)} m_{ef}(x_f')}
$$
The **Sum Product Algorithm**

- We can obtain all marginals by simply doubling the work needed to compute a single marginal.

- After having passed messages from the leaves to an arbitrary root, we pass messages from the root to back to the leaves using the equation:

  
  \[
  m_{ji}(x_i) = \sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}(x_j) \right)
  \]

- We can then use

  \[
  p(x_f \mid x_E) \propto \psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}(x_f)
  \]

  to find the marginal at every node.

- The complete set of messages scales linearly with the size of the tree.
Below are the messages for computing the marginals of $X_1$, $X_2$ and $X_4$. 

$$m_{21}(x_1)$$

$$m_{32}(x_2)$$

$$m_{12}(x_2)$$

$$m_{32}(x_2)$$

$$m_{24}(x_4)$$
Below are the messages for computing all marginals.

The main idea is to reuse the computed messages.
Message-Passing Protocol

- All that is **needed at each query node is the complete set of incoming messages**

- However, there are dependencies to consider: **a message cannot be sent from j to i until messages have been received from all other neighbors of j**

- Think of this as a parallel algorithm: each node has a processor that polls its neighbors, sends each message when all others received

- The messages will flow in from the leaves.
Synchronous Parallel Protocol

- **Protocol:** node $j$ sends a message to neighbor $i$ when (and only when) it has received messages from all other neighbors.

- All incoming messages received by all nodes in $O(2|\mathcal{E}|)$ steps.

✓ Solid arrows: messages passed at a given time
✓ Dashed arrows: messages passed at earlier times
The solid lines are the messages passed at a given time step and the dotted lines are the messages passed at the earlier step.
Messages start to flow from the leaves. When the algorithm terminates, there are two messages for each edge one in each direction. We need to be sure that the algorithm never blocks.
Let us consider another example of accounting for the evidence $x_1 = 1$ for a distribution having the form:

$$P(x_1, x_2, \ldots, x_n; \theta) = \frac{1}{Z(\theta)} \prod_{(i,j) \in E} \psi_{i,j}(x_i, x_j)$$

An easy way to introduce this evidence is by modifying the clique potentials for all edges $(1, j) \in E$ as follows:

$$\psi_{1,j}(x_1, x_j) = 0 \forall x_1 \neq 1$$

We keep all other definitions of $\psi_{i,j}(x_i, x_j)$ with no changes.

Our Bayesian network is then of the form

$$P(x_2, \ldots, x_n; X_1 = 1, \theta) = \frac{1}{Z(\theta)} \prod_{(i,j) \in E} \psi_{i,j}(x_i, x_j)$$
Let us consider the MRF model shown. We want to compute the partition function as well as marginal probabilities.

\[
P(x_1, x_2, \ldots, x_n; \theta) = \frac{1}{Z(\theta)} \sum_{(i, j) \in E} e^{\theta_{ij}(x_i, x_j)} = \frac{1}{Z(\theta)} \prod_{(i, j) \in E} e^{\theta_{ij}(x_i, x_j)} = \frac{1}{Z(\theta)} \prod_{(i, j) \in E} \psi_{i,j}(x_i, x_j)
\]

\[
Z(\theta) = \sum_{x_1, x_2, \ldots, x_n} \prod_{(i, j) \in E} e^{\theta_{ij}(x_i, x_j)}
\]

\[
P(X_i = x; \theta) = \frac{1}{Z(\theta)} \sum_{x_1, x_2, \ldots, x_n} \delta(x_i, x) \prod_{(i, j) \in E} \psi_{i,j}(x_i, x_j)
\]

We pick node 11 as the root node (arbitrary).
We first send messages up through the tree.

\[ m_{i \to j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \prod_{k \in N(i), k \neq j} m_{k \to i}(x_i), \text{ where } N(i) = \{ j : (i, j) \in E \} \]

Note if \( N(i) = \{ j \} \), then the message is simply

\[ m_{i \to j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \]

The particular calculations of these messages are shown next.
Bottom Up Dynamic Programming

\[
m_{1 \rightarrow 7}(x_7) = \sum_{x_1} \psi_{1,7}(x_1, x_7) \quad m_{3 \rightarrow 8}(x_8) = \sum_{x_3} \psi_{3,8}(x_3, x_8) \quad m_{5 \rightarrow 10}(x_{10}) = \sum_{x_5} \psi_{5,10}(x_5, x_{10})
\]

\[
m_{2 \rightarrow 7}(x_7) = \sum_{x_2} \psi_{2,7}(x_2, x_7) \quad m_{4 \rightarrow 8}(x_8) = \sum_{x_4} \psi_{4,8}(x_4, x_8) \quad m_{6 \rightarrow 10}(x_{10}) = \sum_{x_6} \psi_{6,10}(x_6, x_{10})
\]

\[
m_{7 \rightarrow 9}(x_9) = \sum_{x_7} \psi_{7,9}(x_7, x_9) m_{1 \rightarrow 7}(x_7) m_{2 \rightarrow 7}(x_7)
\]

\[
m_{8 \rightarrow 9}(x_9) = \sum_{x_8} \psi_{8,9}(x_8, x_9) m_{3 \rightarrow 8}(x_8) m_{4 \rightarrow 8}(x_8)
\]

\[
m_{9 \rightarrow 11}(x_{11}) = \sum_{x_9} \psi_{9,11}(x_9, x_{11}) m_{7 \rightarrow 9}(x_9) m_{8 \rightarrow 9}(x_9)
\]

\[
m_{10 \rightarrow 11}(x_{11}) = \sum_{x_{10}} \psi_{10,11}(x_{10}, x_{11}) m_{5 \rightarrow 10}(x_{10}) m_{6 \rightarrow 10}(x_{10})
\]

\[
Z(\theta) = \sum_{x_{11}} m_{9 \rightarrow 11}(x_{11}) m_{10 \rightarrow 11}(x_{11})
\]
Let us consider the message $m_{9 \rightarrow 11}(x_{11})$.

We define the subtree $T(9,11)$ – it contains the edge 9,11 and the subtree rooted at node 9 that is the component when the edge (9,11) is removed from the graph.

$$m_{9 \rightarrow 11}(x_{11}) = \sum_{x_1, x_2, x_3, x_4, x_7, x_8, x_9} \prod_{(i,j) \in T(9,11)} \psi_{i,j}(x_i, x_j)$$
We now compute messages send from the root to the leafs. The same equations are applied as before.

\[ m_{i\to j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \prod_{k \in N(i), k \neq j} m_{k\to i}(x_i), \text{ where } N(i) = \{j : (i, j) \in E\} \]

Note if \( N(i) = \{j\} \), then the message is simply

\[ m_{i\to j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \]

The particular calculations of these messages are shown next.
\[ m_{11 \to 9}(x_9) = \sum_{x_{11}} \psi_{11,9}(x_{11}, x_9) m_{10 \to 11}(x_{11}) \]

\[ m_{11 \to 10}(x_{10}) = \sum_{x_{11}} \psi_{11,10}(x_{11}, x_{10}) m_{9 \to 11}(x_{11}) \]

\[ m_{9 \to 7}(x_7) = \sum_{x_9} \psi_{9,7}(x_9, x_7) m_{11 \to 9}(x_9) m_{8 \to 9}(x_9) \]

\[ m_{9 \to 8}(x_8) = \sum_{x_9} \psi_{9,8}(x_9, x_8) m_{11 \to 9}(x_9) m_{7 \to 9}(x_9) \]
Once all messages are computed, one can easily compute all the marginals.

\[ P(X_i = x; \theta) = \frac{1}{Z(\theta)} \prod_{j \in N(i)} m_{j \rightarrow i}(x), \text{ where } N(i) = \{ j : (i, j) \in E \} \]

For example:

\[ P(X_9 = x; \theta) = \frac{1}{Z(\theta)} m_{7 \rightarrow 9}(x)m_{8 \rightarrow 9}(x)m_{11 \rightarrow 9}(x) \]
Consider a typical message $m_{11 \rightarrow 9}(x_9)$.

We consider the subtree $T(11,9)$ that contains the edge 11,9 and the subtree rooted at node 11 that is the component when the edge (11,9) is removed from the graph.

$$m_{11 \rightarrow 9}(x_9) = \sum_{x_5, x_6, x_{10}, x_{11}} \prod_{(i,j) \in T(11,9)} \psi_{i,j}(x_i, x_j)$$
postorder(u)
if u is a leaf
   print u,
else
   postorder(u $\rightarrow$ leftChild);
   postorder(u $\rightarrow$ rightChild);
   print u;
end
**Sum Product Algorithm for a Tree**

**Sum-Product(T,E)**

1. Evidence(E)
   
   \[ f = \text{ChooseRoot}(V) \]
   
   for \( e \in N(f) \)
   
   Collect(f,e)
   
   for \( e \in N(f) \)
   
   Distribute(f,e)

2. for \( i \in V \)
   
   ComputeMarginal(i)

**Collect(i,j)**

for \( k \in N(j) \setminus i \)

Collect(j,k)

SendMessage(j,i)

**Distribute(i,j)**

SendMessage(i,j)

for \( k \in N(j) \setminus i \)

Distribute(j,k)

**Evidence(E)**

for \( i \in E \)

\[ \psi^E(x_i) = \psi(x_i) \delta(x_i, \bar{x}_i) \]

for \( i \notin E \)

\[ \psi^E(x_i) = \psi(x_i) \]

**SendMessage(j,i)**

\[ m_{ji}(x_i) = \sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in V(j) \setminus i} m_{kj}(x_j) \right) \]

**ComputeMarginal(i)**

\[ p(x_i) \propto \psi^E(x_i) \prod_{j \in V(i)} m_{ji}(x_i) \]
**Sum Product Algorithm for a Tree**

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

Choose any root (unspecified here)
Messages flow from the leaves to the root
Messages flow outward from the root to the leaves
Belief Propagation: Sum-Product Algorithm

- The message passing algorithm is thus extended to any tree-structured graph (no loops).
- For each node to find the outgoing message:
  - Form product of incoming messages and local evidence (if e.g. node \( j \) here is connected to an observed node)
  - Marginalize over \( x_j \) to give outgoing message at \( x_j \)
  - One message in each direction across each link
- The algorithm fails if there are loops in the graph

\[
m_{ji}(x_i) = \sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}(x_j) \right)
\]

To implement this, propagate messages from the root node to the leaf node and reversely, and save all messages along each and every edge.
As we have seen in an earlier lecture, Factor Graphs are closely related to directed and undirected graphical models, but start with factorization rather than with CI.

They generalize both directed and undirected graphical models.

A slightly modified sum-product algorithm works for factor trees.

This turns out to be a simple way of extending sum-product to polytrees. Exact Belief propagation is NP hard but in polytrees takes linear time.

Factor Graphs also provide a gateway to factor analysis, probabilistic PCA, Kalman filters, etc.

Notation for Factor Graphs

- Given variables \( \{x_1, \ldots, x_n\} \), let \( \mathcal{C} \) be a (multi)set of subsets of the indices \( \{1, \ldots, n\} \).

- For example, \( \mathcal{C} = \{\{1,3\}, \{3,4\}, \{2,4,5\}, \{1,3\}\} \) for \( \{x_1, \ldots, x_5\} \).

- Let there be a function \( f_s(x_{\mathcal{C}_s}) \) associated with each \( \mathcal{C}_s \in \mathcal{C} \). It is called a factor.

- Let the multivariate function \( f \) be defined:

\[
f(x_1, \ldots, x_n) = \prod_{\mathcal{C}_s \in \mathcal{C}} f_{x_{\mathcal{C}_s}}
\]

- This function need not be a probability distribution, but we will assume it is.
Bipartite Factor Graphs

- The graph is denoted $G(V, F, E)$, with variables $V$, factors $F$, and edges $E$. For example:

$$f(x_1, x_2, x_3, x_4, x_5) = f_a(x_1, x_3) f_b(x_3, x_4) f_c(x_2, x_4, x_5) f_d(x_1, x_3)$$

$$E = \{1,3\}, \{3,4\}, \{2,4,5\}, \{1,3\}$$
Converting to Factor Graphs

\[ \psi(x_1, x_2, x_3) = f_a(x_1, x_2) f_b(x_1, x_3) f_c(x_2, x_3) \]
\[
\psi(x_1, x_2, x_3) = f_a(x_1, x_2)f_b(x_1, x_3)f_c(x_2, x_3)
\]

\[
f_a(x_1, x_2)
\]

\[
\psi(Z_1) = f_a(x_1, x_2)
\]

\[
Z_1 \text{ an indicator variable taking 4 values for each combination of the values of } X_1 \text{ and } X_2 \text{ (assumed binary)}
\]

\[
p(W_1 = 1|x_1, x_2) = f_a(x_1, x_2)
\]

\[
W_1 \text{ binary variable always set to 1}
\]
Factor Trees and Polytrees

- Factor trees are factor graphs that are trees, ignoring the distinction between variable nodes and factor nodes.

- Directed and undirected trees can trivially be represented as factor trees.

- Polytrees can also be represented as factor trees.
We assume that our graph is an undirected tree or a directed tree or polytree, so that the corresponding factor graph has a tree structure. The sum-product algorithm is applied in all of these three cases.

We convert the original graph into a factor graph so that we can deal with both directed and undirected models using the same framework.

Objective:
1) to obtain an efficient, exact inference algorithm for finding marginals;
2) in situations where several marginals are required, to allow computations to be shared efficiently.

Key idea: Distributive Law

\[ ab + ac = a(b + c) \]
We begin by considering the problem of finding the marginal $p(x)$ for particular variable node $x$. We consider factor-graphs with a tree structure.

The marginal is

$$p(x) = \sum_{x \setminus x} p(x)$$

where $x \setminus x$ denotes the set of variables in $x$ with variable $x$ omitted.

The joint distribution:

$$p(x) = \prod_{s \in \text{ne}(x)} F_s(x, X_s)$$
The joint distribution:

\[ p(x) = \prod_{s \in \text{ne}(x)} F_s(x, X_s) \]

- Here ne(x) denotes the set of factor nodes that are neighbors of x.
- \(X_s\) denotes the set of all variables in the subtree connected to the variable node x via the factor node \(f_s\), and
- \(F_s(x, X_s)\) represents the product of all the factors in the group associated with factor \(f_s\).
Messages from Factor Nodes to Variable Nodes

Here we define a function

$$\mu_{f_s \rightarrow x}(x) = \sum_{X_s} F_s(x, X_s)$$

which can be viewed as message from the factor node $f_s$ to the variable node $x$.

We see that the required marginal $p(x)$ is given by the product of all the incoming messages arriving at node $x$. 

$$p(x) = \prod_{s \in \text{ne}(x)} \left[ \sum_{X_s} F_s(x, X_s) \right] = \prod_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x)$$
The Sum-Product Algorithm

\[ \mu_{f_s \rightarrow x}(x) = \sum_{X_s} F_s(x, X_s) \]

Here, we have denoted the variables associated with factor \( f_s \), in addition to \( x \), by \( x_1, \ldots, x_M \).

\[ F_s(x, X_s) = f_s(x, x_1, \ldots, x_M) G_1(x_1, X_{s1}) \cdots G_M(x_M, X_{sM}) \]
The Sum-Product Algorithm

\[ \mu_{f_s \rightarrow x}(x) = \sum_{x_1} \cdots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{s \in \text{ne}(f_s) \setminus \{x\}} \left[ \sum_{X_{x_m}} G_m(x_m, X_{sm}) \right] \]

\[ = \sum_{x_1} \cdots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{s \in \text{ne}(f_s) \setminus \{x\}} \mu_{x_m \rightarrow f_s}(x_m) \]

where \( \text{ne}(f_s) \) denotes the set of variable nodes that are neighbors of the factor node \( f_s \), and \( \text{ne}(f_s) \setminus \{x\} \) denotes the same set but with node \( x \) removed.

\[ \mu_{x_m \rightarrow f_s}(x_m) \equiv \sum_{X_{x_m}} G_m(x_m, X_{sm}) \]

\[ = \sum_{X_{x_m}} \prod_{l \in \text{ne}(x_m) \setminus f_s} F_l(x_m, X_{ml}) \]

\[ = \prod_{l \in \text{ne}(x_m) \setminus f_s} \mu_{f_l \rightarrow x_m}(x_m) \]
Initialization

- In order to start this recursion, we can view the node $x$ as the root of the tree and begin at the leaf nodes.

- The sum-product algorithm begins with messages sent by the leaf nodes, which depend on whether the leaf node is (left) a variable node, or (right) a factor node.

\[
\mu_{x \rightarrow f(x)} = 1
\]

\[
\mu_{f \rightarrow x(x)} = f(x)
\]
The sum-product algorithm can be viewed purely in terms of messages sent out by factor nodes to other factor nodes.

The outgoing message shown by the blue arrow is obtained by

- taking the product of all the incoming messages shown by green arrows,
- multiplying by the factor $f_s$, and
- marginalizing over the variables $x_1$ and $x_2$. 
To compute local marginals:

- Pick an arbitrary node as a root

- Compute and propagate messages from the leaf nodes to the root, storing received messages at every node.

- Compute and propagate messages from the root to the leaf nodes, storing received messages at every node.

- Compute the product of received messages at each node for which the marginal is required, and normalize if necessary.
Applying the Sum-Product Algorithm to a Markov Chain gives as expected the same results as the elimination algorithm using $\alpha$- and $\beta$-messages.

\[
\begin{align*}
\mu_{f_{n-2,n-1} \rightarrow x_{n-1}}(x_{n-1}) & \quad \mu_{f_{n-1,n} \rightarrow x_n}(x_n) & \quad \mu_{f_{n,n+1} \rightarrow x_n}(x_n) & \quad \mu_{f_{n+1,n+2} \rightarrow x_{n+1}}(x_{n+1}) \\
\end{align*}
\]

\[
p(x_n) = \mu_{f_{n-1,n} \rightarrow x_n}(x_n)\mu_{f_{n,n+1} \rightarrow x_n}(x_n) = \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n)\mu_{x_{n-1} \rightarrow f_{n-1,n}}(x_{n-1})\sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1})\mu_{x_{n+1} \rightarrow f_{n,n+1}}(x_{n+1})
\]

\[
= \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \left( \mu_{f_{n-2,n-1} \rightarrow x_{n-1}}(x_{n-1}) \right) \sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1}) \left( \mu_{f_{n+1,n+2} \rightarrow x_{n+1}}(x_{n+1}) \right)
\]

The end nodes are variable nodes so they send unit messages:

\[
\mu_{\alpha}(x) = \sum_{x_1} \psi_{1,2}(x_1, x_2)\mu_{x_1 \rightarrow f_{1,2}}(x_1) = \sum_{x_1} \psi_{1,2}(x_1, x_2)
\]
Example: a simple 4-node factor graph

Un-normalized joint distribution:

\[ p(x) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4) \]
Sum-Product: Example

\[
\begin{align*}
\mu_{x_1 \rightarrow f_a}(x_1) &= 1 \\
\mu_{f_a \rightarrow x_2}(x_2) &= \sum_{x_1} f_a(x_1, x_2) \\
\mu_{x_4 \rightarrow f_c}(x_4) &= 1 \\
\mu_{f_c \rightarrow x_2}(x_2) &= \sum_{x_4} f_c(x_2, x_4) \\
\mu_{x_2 \rightarrow f_b}(x_4) &= \mu_{f_a \rightarrow x_2}(x_2) \mu_{f_c \rightarrow x_2}(x_2) \\
\mu_{f_b \rightarrow x_3}(x_3) &= \sum_{x_2} f_b(x_2, x_3) \mu_{x_2 \rightarrow f_b}(x_2)
\end{align*}
\]
Sum-Product: Example

\[ \mu_{x_3 \rightarrow f_b} (x_3) = 1 \]

\[ \mu_{f_b \rightarrow x_2} (x_2) = \sum_{x_3} f_b (x_2, x_3) \]

\[ \mu_{x_2 \rightarrow f_a} (x_2) = \mu_{f_b \rightarrow x_2} (x_2) \mu_{f_c \rightarrow x_2} (x_2) \]

\[ \mu_{f_a \rightarrow x_1} (x_1) = \sum_{x_2} f_a (x_1, x_2) \mu_{x_2 \rightarrow f_a} (x_2) \]

\[ \mu_{x_2 \rightarrow f_c} (x_2) = \mu_{f_a \rightarrow x_2} (x_2) \mu_{f_b \rightarrow x_2} (x_2) \]

\[ \mu_{f_c \rightarrow x_4} (x_4) = \sum_{x_2} f_c (x_2, x_4) \mu_{x_2 \rightarrow f_c} (x_2) \]
\[ p(x_2) = \mu_{f_a \to x_2}(x_2) \mu_{f_b \to x_2}(x_2) \mu_{f_c \to x_2}(x_2) \]

\[ = \left[ \sum_{x_1} f_a(x_1, x_2) \right] \left[ \sum_{x_3} f_b(x_2, x_3) \right] \left[ \sum_{x_4} f_c(x_2, x_4) \right] \]

\[ = \sum_{x_1} \sum_{x_3} \sum_{x_4} f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4) \]

\[ = \sum_{x_1} \sum_{x_3} \sum_{x_4} p(x) \]
Another example schedule of messages resulting in computing $p(x_4)$

- Initialize all messages to be $1$

<table>
<thead>
<tr>
<th>message direction</th>
<th>message value</th>
<th>If a variable (here $x_1$) has only one factor as a neighbor, it can initiate message propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \rightarrow f_1$</td>
<td>$1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>$x_3 \rightarrow f_2$</td>
<td>$1(x_3)$</td>
<td></td>
</tr>
<tr>
<td>$f_1 \rightarrow x_2$</td>
<td>$\sum_{x_1} f_1(x_1, x_2)1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>$f_2 \rightarrow x_2$</td>
<td>$\sum_{x_3} f_2(x_3, x_2)1(x_3)$</td>
<td></td>
</tr>
<tr>
<td>$x_2 \rightarrow f_3$</td>
<td>$\left(\sum_{x_1} f_1(x_1, x_2)\right)\left(\sum_{x_3} f_2(x_3, x_2)\right)$</td>
<td></td>
</tr>
<tr>
<td>$f_3 \rightarrow x_4$</td>
<td>$\sum_{x_2} f_3(x_2, x_4)\left(\sum_{x_1} f_1(x_1, x_2)\right)\left(\sum_{x_3} f_2(x_3, x_2)\right)$</td>
<td></td>
</tr>
</tbody>
</table>
Once a variable has received all the messages from its neighboring factors, we can compute the probability of that variable by multiplying all the messages and renormalizing:

\[ p(x) \propto \prod_{h \in \text{ne}(x)} \mu_{h \rightarrow x}(x) \]
- Initialize all messages to be 1
- An example schedule of messages resulting in computing $p(x_4|x_1=a)$:

<table>
<thead>
<tr>
<th>message direction</th>
<th>message value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \to f_1$</td>
<td>$\delta(x_1 = a)$</td>
</tr>
<tr>
<td>$x_3 \to f_2$</td>
<td>$1(x_3)$</td>
</tr>
<tr>
<td>$f_1 \to x_2$</td>
<td>$\sum_{x_1} f_1(x_1, x_2) \delta(x_1 = a) = f_1(x_1 = a, x_2)$</td>
</tr>
<tr>
<td>$f_2 \to x_2$</td>
<td>$\sum_{x_3} f_2(x_3, x_2) 1(x_3)$</td>
</tr>
<tr>
<td>$x_2 \to f_3$</td>
<td>$f_1(x_1 = a, x_2) \left( \sum_{x_3} f_2(x_3, x_2) \right)$</td>
</tr>
<tr>
<td>$f_3 \to x_4$</td>
<td>$\sum_{x_2} f_3(x_2, x_4) f_1(x_1 = a, x_2) \left( \sum_{x_3} f_2(x_3, x_2) \right)$</td>
</tr>
</tbody>
</table>
Marginal Associated with Each Factor

The marginal distributions \( p(x_s) \) over the sets of variables \( x_s \) associated with each of the factors \( f_s(x_s) \) in a factor graph can be found by first running the sum-product message passing algorithm and then evaluating the required marginals.

The marginal: 
\[
p(x_s) = \sum_{x \neq x_s} p(x)
\]

\[
p(x_s) \equiv \sum_{x \neq x_s} \prod_{i \in ne(f_s)} \prod_{j \in ne(x_i) \setminus f_s} F_j(x_i, X_{ij})
\]

Product of all the factors in the group associated with factor \( j \) connected to node \( i \). \( X_{ij} \) are all variables on the sub-tree \( ij \).

\[
= f_s(x_s) \prod_{i \in ne(f_s)} \sum_{x \neq x_s} \prod_{j \in ne(x_i) \setminus f_s} F_j(x_i, X_{ij})
\]

\[
= f_s(x_s) \prod_{i \in ne(f_s)} \mu_{x_i \rightarrow f_s}(x_i)
\]
As an application, let us consider computing $p(x_1, x_2)$ in the factor graph below:

$$p(x_s) = f_s(x_s) \prod_{i \in \text{ne}(f_s)} \mu_{x_i \rightarrow f_s}(x_i)$$

$$p(x_1, x_2) = f_a(x_1, x_2) \mu_{x_1 \rightarrow f_a}(x_1) \mu_{x_2 \rightarrow f_a}(x_2)$$

$$= f_a(x_1, x_2) \mu_{x_2 \rightarrow f_a}(x_2)$$

$$= f_a(x_1, x_2) \mu_{f_b \rightarrow x_2}(x_2) \mu_{f_c \rightarrow x_2}(x_2)$$

$$= f_a(x_1, x_2) \sum_{x_3} f_b(x_2, x_3) \sum_{x_4} f_c(x_2, x_4)$$

$$= \sum_{x_3} \sum_{x_4} f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4)$$

$$= \sum_{x_3} \sum_{x_4} p(x)$$
Suppose we want to compute $p(x_a, x_b)$ where the set of variables $x_a$ and $x_b$ do not belong to the same factor.

$$p(x_a, x_b) = p(x_b | x_a) p(x_a)$$

The marginal $p(x_a)$ can be computed by using the sum-product algorithm over all variables including $x_b$.

To compute the conditional $p(x_b | x_a)$, fix the evidence $x_a$ and for each of its allowed values run the sum-product algorithm to compute $p(x_b | x_a)$ by marginalizing over all variables except $x_b$ and $x_a$ (that remains at its fixed value).
The marginal $p(x_i)$ can also be written as the product of the incoming message along any one of the links with the outgoing message along the same link.

\[
p(x_i) \equiv \prod_{s \in \text{ne}(x_i)} \mu_{s \rightarrow x_i}(x_i) \\
= \mu_{f_s \rightarrow x_i}(x_i) \prod_{t \in \text{ne}(x_i) \setminus f_s} \mu_{f_t \rightarrow x_i}(x_i) \\
= \mu_{f_s \rightarrow x_i}(x_i) \mu_{x_i \rightarrow f_s}(x_i)
\]
We often use different notation for the two type of messages:
- Variables $\rightarrow$ factors ($v$)
- Factors $\rightarrow$ Variables ($\mu$)

Both products of incoming messages but only variables require summing:

\[
\begin{align*}
\nu_{is}(x_i) &= \prod_{t \in \mathcal{V}(i) \setminus \{s\}} \mu_{ti}(x_i) \\
\mu_{si}(x_i) &= \sum_{x_s \in \mathcal{X}(s)} \left[ f_s(x_s) \prod_{j \in \mathcal{V}(s) \setminus \{i\}} \nu_{js}(x_j) \right]
\end{align*}
\]
Sum-Product Algorithm for a Factor-Tree

Sum-Product($\mathcal{F}, E$)

Evidence($E$)

\[ f = \text{ChooseRoot}(\mathcal{U}) \]

for $s \in \mathcal{N}(f)$

\[ \mu\text{-Collect}(f,s) \]

\[ \nu\text{-Distribute}(f,s) \]

for $i \in \mathcal{V}$

\[ \text{ComputeMarginal}(i) \]

Evidence($E$)

\[ \text{for } i \in E \]

\[ \psi^E(x_i) = \psi(x_i)\delta(x_i, x_i) \]

\[ \text{for } i \notin E \]

\[ \psi^E(x_i) = \psi(x_i) \]

\[ \mu\text{-Collect}(i,s) \]

\[ \text{for } j \in \mathcal{N}(s) \setminus i \]

\[ \nu\text{-Collect}(s,j) \]

\[ \mu\text{-SendMessage}(s,i) \]

\[ \mu\text{-Distribute}(s,i) \]

\[ \mu\text{-SendMessage}(s,i) \]

\[ \text{for } t \in \mathcal{N}(i) \setminus s \]

\[ \nu\text{-Distribute}(i,t) \]

\[ \nu\text{-SendMessage}(i,s) \]

\[ \nu\text{-Distribute}(i,s) \]

\[ \text{for } j \in \mathcal{N}(s) \setminus i \]

\[ \mu\text{-Distribute}(s,j) \]
Sum Product Algorithm for a Factor-Tree

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

\[ \nu_{is}(x_i) = \prod_{i \in N(i) \setminus \{s\}} \mu_{ti}(x_i) \]

ComputeMarginal(i)

\[ p(x_i) \propto \nu_{is}(x_i) \mu_{si}(x_i) \]
Another Example

\[ \mu_{a1}(x_1) = \sum_{x_{M(a)}(1)} \left( f_a(x_{M(a)}) \prod_{j \in M(a)} \nu_j(x_j) \right) = f_a(x_1) = \psi^E(x_1), \mu_{b2}(x_2) = \psi^E(x_2), \mu_{c3}(x_3) = \psi^E(x_3) \]
Another Example

(a) $X_1 \quad X_2 \quad X_3$

(b) $X_1 \quad f_d \quad X_2 \quad f_e \quad X_3$

(c) $\mu_{a1}(x_1) \quad \mu_{b2}(x_2) \quad \mu_{c3}(x_3)$

(d) $v_{1d}(x_1) \quad v_{2e}(x_3)$

(e) $\mu_{d2}(x_2) \quad \mu_{e2}(x_2)$

(f) $v_{2d}(x_2) \quad v_{2e}(x_2)$

(g) $\mu_{d1}(x_1) \quad \mu_{e3}(x_3)$

(h) $v_{1d}(x_1) \quad v_{3e}(x_3)$
Example

\[ \nu_{1d}(x_1) = \prod_{t \in \mathcal{F}(1) \setminus \{d\}} \mu_{t1}(x_1) = \mu_{d1}(x_1) = \psi^E(x_1), \nu_{3e}(x_3) = \psi^E(x_3) \]

\[ \mu_{d2}(x_2) = \sum_{x_{\mathcal{F}(d) \setminus \{d\}}} \left( f_d(x_{\mathcal{F}(d)}) \prod_{j \in \mathcal{F}(d) \setminus \{2\}} \nu_{jd}(x_j) \right) = \sum_{x_1} \psi(x_1, x_2) \psi^E(x_1), \mu_{e2}(x_2) = \sum_{x_3} \psi(x_2, x_3) \psi^E(x_3) \]
Example

\[ \nu_{2d}(x_2) = \prod_{i \in V(2) \{d\}} \mu_{t2}(x_2) = \psi^E(x_2) \sum_{x_3} \psi^E(x_3) \psi(x_2, x_3), \]
\[ \nu_{2e}(x_2) = \psi^E(x_2) \sum_{x_1} \psi^E(x_1) \psi(x_1, x_2) \]
\[ \nu_{2b}(x_2) = \sum_{x_1} \psi^E(x_1) \psi(x_1, x_2) \sum_{x_3} \psi^E(x_3) \psi(x_2, x_3) \]

\[ \mu_{d1}(x_1) = \sum_{x_{d'(d)\{1\}}} \left( f_d(x_{d'(d)}) \prod_{j \in V(d) \{1\}} \nu_{jd}(x_j) \right) = \sum_{x_2} \psi^E(x_2) \psi(x_1, x_2) \sum_{x_3} \psi^E(x_3) \psi(x_2, x_3) = \nu_{1a}(x_1) \]
\[ \mu_{e3}(x_3) = \sum_{x_2} \psi^E(x_1) \psi(x_1, x_2) \sum_{x_3} \psi^E(x_2) \psi(x_2, x_3) = \nu_{3c}(x_3) \]

Note that these messages are the same as the corresponding messages that would pass in a run of the SUM-PRODUCT algorithm in the corresponding undirected graph, e.g.
\[ \mu_{d1}(x_1) = m_{21}(x_1), \mu_{e3}(x_3) = m_{23}(x_3) \]
Converting an undirected graph to a factor graph gives $m$-messages that are the same as the $m$ messages obtained from the SUM-PRODUCT algorithm.

\[
\mu_{si}(x_i) = \sum_{x_{\mathcal{N}(s)\backslash\{i\}}} \left( f_s(x_{\mathcal{N}(s)}) \prod_{j \in \mathcal{N}(s)\backslash\{i\}} \nu_{js}(x_j) \right) = \sum_{x_j} \psi(x_i, x_j) \nu_{js}(x_j) = \sum_{x_j} \psi(x_i, x_j) \prod_{t \in \mathcal{N}(j)\backslash\{s\}} \mu_{ij}(x_j)
\]

\[
= \sum_{x_j} \psi(x_i, x_j) \psi^E(x_j) \prod_{t \in \mathcal{N}(j)\backslash\{s\}} \mu_{ij}(x_j)
\]

In the $\mathcal{N}'(j)$ omitting the singleton factor.
If a graph (directed or undirected) is originally a tree, there is little to gain by transforming it to factor graph.

However, there is significant merit in transforming to factor graphs \textit{various `tree-like' graphs}. The SUM-PRODUCT applies directly to factor trees.

\[ p(x) \propto \psi(x_1, x_2)\psi(x_3, x_5)\psi(x_4, x_6)\psi(x_2, x_3, x_4) \]

(a) An undirected graph with an unfactorized potential \( \psi(x_2, x_3, x_4) \)

(b) An equivalent undirected model using a super variable \( Z \) (range the Cartesian product of the range of \( X_2, X_3, X_4 \)). Create new potentials \( \psi(x_1, Z)\psi(x_5, Z)\psi(x_6, Z)\psi(Z) \)

(c) An equivalent factor graph that is a factor-tree. No need for new potentials.
SUM-PRODUCT is Applied to Factor Trees

- If the variables in the undirected graph can be clustered in non-overlapping cliques, and the parametrization of each clique is a general non-factorized potential, then the corresponding factor graph is a tree and the SUM-PRODUCT algorithm applies.

\[ p(x) \propto \psi(x_1, x_2)\psi(x_3, x_5)\psi(x_4, x_6)\psi(x_2, x_3, x_4) \]
Recall that a polytree is a directed graph that reduces to an undirected tree if we convert each directed edge to an undirected edge. Polytrees have no loops in their underlying undirected graph.

The corresponding factor graph is a tree with a factor for each family $p(x_i|x_{\pi(i)})$ and the SUM-PRODUCT algorithm applies.
Inference in HMM involves one forward and one backward pass:

\[
P(Y_1 = y_1, ..., Y_m = y_m, X_1 = x_1, ..., X_m = x_m) = 
P(X_1 = x_1) \prod_{j=2}^{m} P(X_j = x_j | X_{j-1} = x_{j-1}) \prod_{j=1}^{m} P(Y_j = y_j | X_j = x_j)
\]

The computational cost grows linearly with the length of the chain.

Similarly for the Kalman Filter.
Belief propagation is an algorithm for exact inference on directed graphs without loops and is equivalent to a special case of the sum-product algorithm.

We briefly reviewed next for historical reasons (note the sum-product algorithm is more general).

Consider the graph shown which represents:

\[ p(A,B,C,D,E) = p(A)p(B)p(C|A,B)p(D|B,C)p(E|C,D) \]

Inference: evaluate the probability distribution over some set of variables, given the values of another set of variables.

For example, how can we compute \( P(A|C = c) \)? Assume each variable is binary.

**Naive method (total 20 operations):**

\[
p(A, C = c) = \sum_{B,D,E} p(A, B, C = c, D, E) \quad \text{[16 operations, 8 sums on B, D, E for each of the two values of A]}
\]

\[
p(C = c) = \sum_A p(A, C = c) \quad \text{[2 operations]}
\]

\[
p(A|C = c) = \frac{p(A, C = c)}{p(C = c)} \quad \text{[2 operations one for each of the values of A]}
\]
Consider the right graph which represents:

\[ p(A, B, C, D, E) = p(A)p(B)p(C|A, B)p(D|B, C)p(E|C, D) \]

Computing \( p(A|C = c) \).

More efficient method:

\[
p(A, C = c) = \sum_{B,D,E} p(A)p(B)p(C = c|A,B)p(D|B,C = c)p(E|C = c,D)
\]

\[
= \sum_{B} p(A)p(B)p(C = c|A,B) \sum_{D} p(D|B,C = c) \sum_{E} p(E|C = c,D)
\]

\[
= \sum_{B} p(A)p(B)p(C = c|A,B) \quad [4 \text{ operations, sum on 2 values of } B \text{ for each of the 2 values of } A]
\]

Total: 4+2+2 = 8 operations

Belief propagation methods use the conditional independence relationships in a graph to do efficient inference (for singly connected graphs, exponential gains in efficiency!).
A DAG is singly connected if its underlying undirected graph is a tree, i.e. there is only one undirected path between any two nodes.

For some node $X$, we want to compute $p(X|e)$ given evidence $e$.

Since we are considering singly connected graphs:
- every node $X$ divides the evidence into upstream $e_X^+$ from $X$ and downstream $e_X^-$ from $X$
- every edge $X \rightarrow Y$ divides the evidence into upstream $e_{XY}^+$ from the $XY$ edge and downstream $e_{XY}^-$ from $XY$. 
Idea 1: The probability of a variable $X$ can be found by combining upstream and downstream evidence:

$$p(X|e) = \frac{p(X,e)}{p(e)} = \frac{p(X,e^+_X,e^-_X)}{p(e^+_X,e^-_X)} \propto p(X|e^+_X) \times p(e^-_X|X,e^+_X)$$

$$= p(X|e^+_X)p(e^-_X|X) = \pi(X) \lambda(X)$$

Idea 2: The upstream and downstream evidence can be computed via a local message passing algorithm between the nodes in the graph.

Idea 3: “Don’t send back to a node (any part of) the message it sent to you!”
Belief Propagation in Directed Graphs

\[ p(X|e) = \frac{p(X|e^+_U) p(e^-_X|X)}{\pi(X)} \lambda(x) \]

Top-down upstream evidence: message \( U_i \) (parent) sends to \( X \)

\[ \pi_x(U_i) = p(U_i|e^+_{U_i,X}) \quad \text{Function of } U_i \]

bottom-up downstream evidence: message \( Y_j \) (child) sends to \( X \)

\[ \lambda_y(X) = p(e^-_{XY_j}|X) \quad \text{Function of } X \]

update the probability of \( X \) given the evidence:

Belief of \( X \): \( \text{BEL}(X) = p(X|e) = \frac{1}{Z} \lambda(X) \pi(X) \)

info from the \( \text{info from the} \)

parents of \( X \)

children of \( X \)

Bottom up Belief State: All the information
the children of \( X \) send to \( X \)

\[ \lambda(X) = \prod_j \lambda_{y_j}(X) \]

Conditional between \( X \) and all of its parents
All the information the parents of \( X \) send to \( X \)

\[ \pi(X) = \sum_{U_1,...,U_n} p(X|U_1,...,U_n) \prod_i \pi_x(U_i) \]
Messages to Pass

\( \lambda(X) = \prod_j \lambda_j(X) \)

\( \pi(X) = \sum_{U_1,\ldots,U_n} p(X|U_1,\ldots,U_n) \prod_i \pi_X(U_i) \)

**Top-down propagation, message X sends to Y_j:**

\[ \pi_{Y_j}(X) = \frac{1}{Z} \left[ \prod_{k \neq j} \lambda_k(X) \right] \sum_{U_1,\ldots,U_n} p(X|U_1,\ldots,U_n) \prod_i \pi_X(U_i) = \frac{1}{Z} \frac{\text{BEL}(X)}{\lambda_{Y_j}(X)} \]

\( Z \) is the normalization constant ensuring: \( \sum_X \pi_{Y_j}(X) = 1 \)

**Bottom-up propagation, message X sends to U_i:**

\[ \lambda_{X}(U_i) = \sum_X \lambda(X) \sum_{U_{k: k \neq i}} p(X|U_1,\ldots,U_n) \prod_{k \neq i} \pi_X(U_k) \]

You send back all messages received from your children and all your parents (except \( U_i \)).

You send the information received from your children (except \( Y_j \)) and all your parents.

**top-down upstream evidence:**
message \( U_i \) sends to \( X \)

\[ \pi_X(U_i) = p(U_i|e^+_{U_iX}) \]

**bottom-up downstream evidence:**
message \( Y_j \) sends to \( X \)

\[ \lambda_{Y_j}(X) = p(e^-_{XY_j}|X) \]

Objective: an efficient algorithm for finding

i. the value $x^{\text{max}}$ that maximizes $p(x)$;
ii. the value of $p(x^{\text{max}})$.

In general, maximum marginals $\neq$ joint maximum.

<table>
<thead>
<tr>
<th></th>
<th>$x = 0$</th>
<th>$x = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = 0$</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>$y = 1$</td>
<td>0.3</td>
<td>0.0</td>
</tr>
</tbody>
</table>

$p(x=0) = 0.6$ and $p(x=1) = 0.4$ \quad \arg\max_x p(x) = 0$

$p(y=0) = 0.7$ and $p(y=1) = 0.3$ \quad \arg\max_y p(y) = 0$

While for the joint distribution \quad \arg\max_{x,y} p(x, y) = \{1, 0\}
Example

- Let \( x = \{x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n\} \)

- Suppose the marginal probability of \( x_i = a \) is large (maximal for \( x_i \)):
  \[
a = \arg \max_{x_i} p_{x_i} = \arg \max_{x_i} \sum_{y=x\setminus\{x_i\}} p_{x_i, y}
  \]

- Similarly, suppose the marginal probability of \( x_j = b \) is large (maximal for \( x_j \)):
  \[
b = \arg \max_{x_j} p_{x_j} = \arg \max_{x_j} \sum_{y=x\setminus\{x_j\}} p_{x_j, y}
  \]

- However, suppose that configurations with both \( x_i = a \) and \( x_j = b \) are extremely unlikely:
  \[
p_{x_i = a, x_j = b} = \sum_{y=x\setminus\{x_i, x_j\}} p_{x_i = a, x_j = b, y} \approx 0
  \]
The Max-Sum Algorithm

- Suppose we wish to find a setting of all variables that maximizes the joint pdf

- We want to condition on certain “evidence” nodes

- This is closely related to general probabilistic inference but with sum operators replaced by max operators

- The Max-Sum Algorithm can be seen as an application of Dynamic Programming to graphical models.

---

Inference

\[ p \ x_F \mid x_E = \frac{\sum_{x_R} p \ x_E, x_F, x_R}{\sum_{x_R, x_F} p \ x_E, x_F, x_R} \]
We see that conditioning plays little role here. We thus treat the general problem of maximizing a nonnegative factorized function of $n$ variables.

\[ \hat{x}_F = \arg \max_{x_F} p \ x_F \mid x_E = \arg \max_{x_F} p \ x_F, x_E = \arg \max_{x_F} p^E \ x \]
Applications

- Makes sense when a single best solution is desired, e.g,
  - “Parsing” with HMM or CRF (gene prediction, speech recognition, etc.)
  - “Smoothing” with Kalman filter

- The predicted solution will be consistent in a way that a solution built from local marginals might not be
Max Problem Vs. ArgMax Problem

- **Max Problem**: find the maximum value of the pdf

- **Argmax Problem**: find an assignment of the variables that maximizes the pdf (there may be many)

- argmax is of greater interest, but the two problems are closely related

- We begin with max then will solve argmax by extension
Distribution Property

- **Max distributes over multiplication** in the same way that summation does
  
  \[ a \cdot b + a \cdot c = a \cdot (b + c) = a \sum (b, c) \]
  
  \[ \max a \cdot b, a \cdot c = a \max (b, c) \]

- Hence, a max can slide down along a product just as a summation can be,

  \[ \max \max \max f x_1 g x_2 h x_3 = \max f x_1 \max g x_2 \max h x_3 \]

- **Max is also commutative and associative.**

- **Max forms a commutative semi-ring with multiplication**
Max-Product Over A Chain

- Maximizing over a chain (max-product)

\[
p(x^{\text{max}}) = \max_x p(x) = \max_{x_1} \cdots \max_{x_N} p(x)
\]

\[
= \frac{1}{Z} \max_{x_1} \cdots \max_{x_N} \left[ \psi_{1,2}(x_1, x_2) \cdots \psi_{N-1,N}(x_{N-1}, x_N) \right]
\]

\[
= \frac{1}{Z} \max_{x_1} \left[ \max_{x_2} \left[ \psi_{1,2}(x_1, x_2) \left[ \cdots \max_{x_N} \psi_{N-1,N}(x_{N-1}, x_N) \right] \right] \right]
\]
Elimination Algorithm Example

\[
\max p \ x_1, \ldots, x_3, \bar{x}_6 = \max_{x_1} \max_{x_2} \max_{x_3} \max_{x_4} \max_{x_5} \max_{x_6} \max p \ x_1 \ p \ x_2 \ | \ x_1 \ p \ x_3 \ | \ x_1 \ p \ x_4 \ | \ x_2 \ p \ x_5 \ | \ x_3 \ p \ x_6 \ | \ x_2, x_3 \ \delta \ x_6, \bar{x}_6
\]

\[
= \max_{x_1} \ p \ x_1 \ \max_{x_2} \ p \ x_2 \ | \ x_1 \ \max_{x_3} \ p \ x_3 \ | \ x_1 \ \max_{x_4} \ p \ x_4 \ | \ x_2 \ \max_{x_5} \ p \ x_5 \ | \ x_3 \ \max_{x_6} \ p \ x_6 \ | \ x_2, x_3 \ \delta \ x_6, \bar{x}_6
\]

\[
= \max_{x_1} \ p \ x_1 \ \max_{x_2} \ p \ x_2 \ | \ x_1 \ \max_{x_3} \ p \ x_3 \ | \ x_1 \ \max_{x_4} \ p \ x_4 \ | \ x_2 \ \max_{x_5} \ p \ x_5 \ | \ x_3 \ h_6 \ x_2, x_5
\]

\[
= \max_{x_1} \ p \ x_1 \ \max_{x_2} \ p \ x_2 \ | \ x_1 \ h_4 \ x_2 \ | \ x_3 \ \max_{x_4} \ p \ x_3 \ | \ x_1 \ h_3 \ x_2, x_3
\]

\[
= \max_{x_1} \ p \ x_1 \ h_4 \ x_2 \ | \ x_1 \ h_3 \ x_2 \ | \ x_1, x_2
\]

\[
= \max_{x_1} \ p \ x_1 \ h_2 \ x_1 = h_1.
\]
MAP-ELIMINATE Algorithm

MAP-Eliminate(G,E)
  Initialize(G)
  Evidence(E)
  Update(G)
  Maximum

Initialize(G)
  choose an ordering I
  for each node X_i in V
    place p(x_i | x_{π_i}) on the active list
  end

Evidence(E)
  for each node X_i in X_E
    place δ(x_i, x_i) on the active list
  end
MAP-ELIMINATE Algorithm

MAP-Eliminate(\mathcal{G}, E)

Initialize(\mathcal{G})
Evidence(E)
Update(\mathcal{G})
Maximum

- Use logs of potentials and replace "product" with "sum"

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 \( \phi_{i}^{\text{max}}(x_{T_i}) \) be the product of these potentials
let \( m_{i}^{\text{max}}(x_{S_i}) = \max_{x_i} \phi_{i}^{\text{max}}(x_{T_i}) \)
place \( m_{i}^{\text{max}}(x_{S_i}) \) on the active list
end

Maximum

\[ \max_{x} p^{E}(x) = \text{the scalar value on the active list} \]

- After the final node has been eliminated in UPDATE, the active list contains a single scalar value which is the max obtained by the algorithm
Traceback

- When finding the max, only one pass is needed—collect but not distribute.

- For the argmax, however, a distribute step is required.

- It can be thought of as a traceback from the termination node to the leaves, following back-pointers maintained during collect.

- These pointers indicate which variable selections at neighboring nodes permit each candidate solution at a given node.
Max-Product($\mathcal{I}, E$)

Evidence($E$)

for $e \in \mathcal{N}(f)$

Collect($f,e$)

\[
MAP = \max_{x_f} \left\{ \psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}^{\text{max}}(x_f) \right\}
\]

\[
x_f^* = \arg \max_{x_f} \left\{ \psi^E(x_f) \prod_{e \in \mathcal{N}(f)} m_{ef}^{\text{max}}(x_f) \right\}
\]

for $e \in \mathcal{N}(f)$

Distribute($f,e$)

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 $k \in \mathcal{N}(j) \setminus i$

Collect($j,k$)

SendMessage($j,i$)
The Max Product Algorithm

SendMessage\((j,i)\)

\[ m_{ji}^{\text{max}}(x_i) = \max_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}^{\text{max}}(x_j) \right) \]

\[ \delta_{ji}(x_i) = \arg \max_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}^{\text{max}}(x_j) \right) \]

- Suppose that during the inward pass (from the leaves to the root), we maintain a record \( \delta_{ji}(x_i) \) of the maximizing values of nodes when we compute \( m_{ji}^{\text{max}}(x_i) \) (for each \( x_i \), this picks out the value of \( x_j \) that achieves the maximum). We can use this to define a consistent maximizing configuration on an outward pass (from the root to the leaves).

- Start at the root and select \( x_j^* \). Pass this to the children of \( f \) and select \( x_e^* = \delta_{ef}(x_j^*), e \in \mathcal{N}(f) \).

- Continue until you reach the leaves.
Max-Product Algorithm

- As with inference, the new potentials can be thought of as messages, which can be passed among nodes of a (poly)tree.

- For an arbitrary undirected graph the message from $j$ to $i$ is given by,

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

- The final solution can be obtained at a final node $i$ as,

$$\max_{x} p^E(x) \propto \max_{x_i} \left( \psi^E(x_i) \prod_{j \in \mathcal{N}(i)} m_{ji}(x_i) \right)$$
The Max-Sum Algorithm

- Generalizes to tree-structured factor graph
  \[ \max_{x} p(x) = \max_{x_n} \prod_{f_s \in \text{ne}(x_n)} \max_{X_s} f_s(x_n, X_s) \]
  maximizing as close to the leaf nodes as possible

- Max-Product → Max-Sum
  
  - For numerical reasons, use
    \[ \ln \left( \max_{x} p(x) \right) = \max_{x} \ln p(x) \]
  
  - Again, use distributive law
    \[ \max(a + b, a + c) = a + \max(b, c) \]
We thus prevent numerical underflow by performing all computations in log space noticing that:

$$\arg \max_{x_F} p^E(x) = \arg \max_{x_F} \ln p^E(x)$$

Multiplications are now replaced by additions:

$$h_{ji}^\text{max}(x_i) = \max_{x_j} \left( \log \psi^E(x_j) + \log \psi(x_i, x_j) + \sum_{k \in \mathcal{N}(j) \setminus i} h_{kj}^\text{max}(x_j) \right)$$

Because max and addition also form a commutative semi-ring, the algorithm remains unchanged (now called MAX-SUM algorithm Vs. the MAX-PRODUCT algorithm).
Consider computing the following:

\[ m_{ji}(x_i) = \sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{kj}(x_j) \right) \]

Using Log-Messages, we can write the following trick:

\[ h_{ji}(x_i) = \log \left\{ \sum_{x_j} \left( \psi^E(x_j) \psi(x_i, x_j) \exp \left( \sum_{k \in \mathcal{N}(j) \setminus i} h_{kj}(x_j) \right) \right) \right\} \]

\[ = \log \left\{ \sum_j \left( \exp \left( \log \psi^E(x_j) + \log \psi(x_i, x_j) \sum_{k \in \mathcal{N}(j) \setminus i} h_{kj}(x_j) \right) \right) \right\} \]

\[ = \log \left\{ \sum_j \exp(\ell_{ij}) \right\} = \log \left\{ \exp(\ell_{iz}) \left( 1 + \sum_{j: j \neq z} \exp(\ell_{ij} - \ell_{iz}) \right) \right\} \quad (z = \arg \max_j \ell_{ij}) \]

\[ = \ell_{iz} + \log \left\{ 1 + \sum_{j: j \neq z} \exp(\ell_{ij} - \ell_{iz}) \right\} \]
The Max-Sum Algorithm

- **Initialization (leaf nodes)**
  \[ \mu_{x \rightarrow f}(x) = 0 \]
  \[ \mu_{f \rightarrow x}(x) = \ln f(x) \]

- **Recursion**

  Recall: \[ \mu_{f_s \rightarrow x}(x) = \sum_{x_1} \cdots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \] and

  \[ \mu_{x_m \rightarrow f_s}(x_m) = \prod_{l \in \text{ne}(x_m) \setminus f_s} \mu_{f_l \rightarrow x_m}(x_m) \]

  \[ \mu_{f \rightarrow x}(x) = \max_{x_1, \ldots, x_M} \left[ \ln f(x, x_1, \ldots, x_M) + \sum_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f}(x_m) \right] \]

  \[ \mu_{x \rightarrow f}(x) = \sum_{l \in \text{ne}(x) \setminus f} \mu_{f_l \rightarrow x}(x) \]

- **At the root node, the max probability can be computed as:**

  \[ p^{\text{max}} = \max_x \left[ \sum_{s \in \text{ne}(x) \setminus x} \mu_{f_s \rightarrow x}(x) \right] \]
The Max-Sum Algorithm

- **Termination (root node)**

\[
p^\text{max} = \max_x \left[ \sum_{s \in \text{ne}(x)} \mu_{fs \rightarrow x}(x) \right]
\]

\[
x^\text{max} = \arg \max_x \left[ \sum_{s \in \text{ne}(x)} \mu_{fs \rightarrow x}(x) \right]
\]

- There may be multiple configurations of \( x \) all of which give rise to the maximum value for \( p(x) \).

- The message passing algorithm can fail because it is possible for the individual variable values obtained by maximizing the product of messages at each node to belong to different maximizing configurations, giving an overall configuration that no longer corresponds to a maximum.
Consider the particular graph above and propagate messages from the leaf (node 1) to the root (node N).

Suppose we take $x_N$ to be the root node, then in the 1st phase

$$\mu_{x_n \rightarrow f_{n+1}, n} (x_n) = \mu_{f_{n-1}, n \rightarrow x_n} (x_n)$$

$$\mu_{f_{n-1}, n \rightarrow x_n} (x_n) = \arg \max_{x_{n-1}} \left[ \ln f_{n-1, n} (x_{n-1}, x_n) + \mu_{x_{n-1} \rightarrow f_{n-1}, n} (x_n) \right]$$

The initial message sent from the leaf node is simply $\mu_{x_1 \rightarrow f_{1,2}} (x_1) = 0$

The most probable value for $x_N$ is

$$x_N^{\text{max}} = \arg \max_{x_N} \left[ \mu_{f_{N-1, N} \rightarrow x_N} (x_N) \right]$$
To determine the states in previous variables that correspond to the maximizing configuration, we keep track of which values of the variables gave rise to the max state in each variable, i.e. store:

\[ \phi(x_n) = \arg\max_{x_{n-1}} \left[ \ln f_{n-1,n}(x_{n-1}, x_n) + \mu_{x_{n-1} \rightarrow f_{n-1,n}}(x_{n-1}) + \mu_{f_{n-1,n} \rightarrow x_n}(x_n) \right] \]

\[ x_{n-1}^{\max} = \phi(x_n^{\max}) \]

For each state of \( x_n \) there is a unique state of the previous variable \( x_{n-1} \) that maximizes the probability.

Once we know the most probable value of \( x_N \), we can compute the most probable state \( x_{N-1} \) and so on until we reach node \( x_1 \).
Example: Markov chain

For the two values of $x_n$ shown, we see by backtracking the two maximizing paths.

If we had maximize each marginal separately, then we could find ourselves selecting some states from one path and some from the other.
In HMMs, the states $X_t$ are discrete.

In linear Gaussian SSMs, the states are real Gaussian vectors.

Both HMMs and SSMs can be represented as singly connected DAGs.

The forward-backward algorithm in hidden Markov models (HMMs), and the Kalman smoothing algorithm in SSMs are both instances of belief propagation / factor graph propagation.
Application in State Space Models

- For HMM, the max-product algorithm is the ‘Viterbi’ algorithm.

![Diagram showing hidden variables and observed variables in a state space model]

Hidden Variables: $X_i$

Observed Variables: $Y_i$
Cutset Conditioning or “reasoning by assumptions”: Find a small set of variables which, if they were given (i.e. known) would render the remaining graph singly connected.

For each value of these variables run belief propagation on the singly connected network.

Average the resulting beliefs with the appropriate weights (given by normalizing constants).
Inference in Multiple Connected DAGs

- The Junction Tree Algorithm:
  - Form an undirected graph from your directed graph such that no additional conditional independence relationships have been created ("moralization").
  - Lump variables in cliques together and form a tree of cliques - this may require "triangulation". Do inference in this tree of cliques.
  - The cost is exponential in the size of the largest clique
  - Many interesting models have intractably large cliques

- Loopy Belief Propagation: just use BP although there are loops.
  - In this case the terms "upstream" and "downstream" are not clearly defined.
  - No guarantee of convergence except for certain special graphs.
  - But often works well in practice.
The Junction Tree Algorithm

- The sum product algorithm is an exact inference algorithm for general graphs with tree structure (no loops).

- The junction tree algorithm is an efficient exact algorithm for general graphs. Works for both directed and undirected graphs.

- Convert original graph into a tree of cliques.

- Junction tree: the tree is condensed, so that any clique that is a subset of another clique is absorbed into the larger clique.
  
  - Works by turning the initial graph into a junction tree and then running a sum-product-like algorithm.
  
  - The cost is exponential in the size of the largest clique. Intractable on graphs with large cliques.

We are starting with a DAG.

We convert it to an undirected graph (MRF) by moralization as shown on the right (if starting from an undirected graph this step is not required).

We will next group the variables into clusters so that the clusters form a tree.
Finding Variable Clusters

We find the clusters by simulating variable elimination

- Step 1: Select an elimination order
- Step 2: Eliminate variables one by one, connecting the neighbors of each variable to be eliminated, introduce new edges when necessary
- Step 3: track the cliques that are formed in the elimination process

These operations are entirely graphical
We first eliminate node 1.

Its neighbors 2 and 3 are connected.

The first clique is defined as $c_1 = \{1, 2, 3\}$.
We next eliminate node 2.

Its neighbors 3 and 4 are already connected.

The second clique is defined as $c_2 = \{2, 3, 4\}$
Variable Elimination and Cliques

- We next eliminate node 3.
- Its neighbors 4 and 5 are already connected.
- The third clique is defined as $c_3 = \{3, 4, 5\}$

$c_1 = \{1, 2, 3\}$
$c_2 = \{2, 3, 4\}$
$c_3 = \{3, 4, 5\}$
We next eliminate node 4.

The fourth clique is defined as $c_4 = \{4, 5\}$.
Variable Elimination and Cliques

\[ c_1 = \{1, 2, 3\} \]
\[ c_2 = \{2, 3, 4\} \]
\[ c_3 = \{3, 4, 5\} \]
\[ c_4 = \{4, 5\} \]
\[ c_5 = \{5\} \]

- The last node to eliminate is node 5.
- The fifth clique is defined as \( c_5 = \{5\} \)
We drop the cliques that are subsets of other cliques, i.e. we only keep the maximal cliques.

We are now ready to form the clique graph using the maximal cliques $c_1, c_2$, and $c_3$. 

\[ c_1 = \{1, 2, 3\} \]
\[ c_2 = \{2, 3, 4\} \]
\[ c_3 = \{3, 4, 5\} \]
\[ c_4 = \{4, 5\} \]
\[ c_5 = \{5\} \]
We now arrange the resulting maximal cliques into a clique graph.

We consider an edge between any two cliques that have variables in common.
Essentially by the elimination process, the graph is triangulated, which involves finding chord-less cycles containing four or more nodes and adding extra links to eliminate such chord-less cycles.

For instance, the cycle A-C-B-D-A is chord-less a link could be added between A and B or alternatively between C and D.

The joint distribution for the resulting triangulated graph is still defined by a product of the same potential functions, but these are now considered to be functions over expanded sets of variables.

The triangulated graph was used to construct a new tree-structured undirected graph called a join tree, whose nodes correspond to the maximal cliques of the triangulated graph, and whose links connect pairs of cliques that have variables in common.
Assigning Weights to the Edges of the Clique Graph

- A weight is assigned to each edge according to the number of shared variables

- The junction tree is then found as the maximum weight spanning tree over the cliques
The selection of which pairs of cliques to connect in this way is important and is done so as to give a maximal spanning tree.

Of all possible trees that link up the cliques, the one that is chosen is one for which the weight of the tree is largest, where the weight for a link is the number of nodes shared by the two cliques it connects, and the weight for the tree is the sum of the weights for the links.

If the tree is condensed, so that any clique that is a subset of another clique is absorbed into the larger clique, this gives a junction tree.

As a consequence of the triangulation step, the resulting tree satisfies the running intersection property, which means that if a variable is contained in two cliques, then it must also be contained in every clique on the path that connects them. This ensures that inference about variables will be consistent across the graph.
The junction tree satisfies an important *running intersection* property: if any two non-adjacent cliques have variables in common, then the variables also appear along the path between them.

The *property ensures consistency of beliefs about the variables that appear in multiple cliques*.
Finally it is useful to identify *separators (overlaps)* between the adjacent cliques in the tree.

The messages we send in the junction tree only depend on the variables in the separators.
There are often many (equivalent) ways of associating the original distribution with the clique potentials in the junction tree.

We can place the terms in the original distribution in any clique that involves the same set of variables.
There are often many (equivalent) ways of associating the original distribution with the clique potentials in the junction tree.

We can place the terms in the original distribution in any clique that involves the same set of variables.
Once we have the clique potentials, we can run the belief propagation algorithm in the junction tree.

Each message depends only on the variables in the separators. It can be evaluated at the cost of summing over the variables in each clique, e.g.

$$m_{1 \to 2}(x_2, x_3) = \sum_{x_1} \psi_1(x_1, x_2, x_3) \forall x_2, x_3$$
Finally, a two-stage message passing algorithm (essentially the sum-product algorithm) can be applied to this junction tree in order to find marginals and conditionals.

At the heart of the junction tree algorithm is the simple idea of exploiting the factorization properties of the distribution to allow sums and products to be interchanged so that partial summations can be performed, thereby avoiding having to work directly with the joint distribution.

The role of the junction tree is to provide a precise and efficient way to organize these computations. This is achieved using purely graphical operations!

The junction tree is exact for arbitrary graphs and is efficient in the sense that for a given graph there does not in general exist a computationally cheaper approach.
Belief Propagation: Distribute Away from the Root

- Once we have the clique potentials, we can run the belief propagation algorithm in the junction tree.

- Each message depends only on the variables in the separators and can be evaluated at the cost of summing over the variables in each clique.

\[ m_{2\rightarrow 1}(x_2, x_3) = \sum_{x_4} \psi_2(x_2, x_3, x_4) m_{3\rightarrow 2}(x_3, x_4) \quad \forall \ x_2, x_3 \]
Belief Propagation: Marginals

- We now have the probabilities of interest stored in the clique potentials and the incoming messages:

\[
P(x_1, x_2, x_3) \propto \psi_1(x_1, x_2, x_3) m_{2 \rightarrow 1}(x_2, x_3)
\]

\[
P(x_2, x_3, x_4) \propto \psi_2(x_2, x_3, x_4) m_{1 \rightarrow 2}(x_2, x_3) m_{3 \rightarrow 2}(x_3, x_4)
\]
The complexity of inference in the junction tree depends on the size of the cliques.

The cliques are determined by our choice of the elimination order (triangulation). Finding the optimal order is hard.

Myopic heuristics often work well: eliminate next the variable that results in the smallest clique size.
The algorithm works with the joint distributions within each node (each of which corresponds to a clique of the triangulated graph) and so the computational cost of the algorithm is determined by the number of variables in the largest clique and will grow exponentially with this number in the case of discrete variables.

An important concept is the treewidth of a graph, which is defined in terms of the number of variables in the largest clique. In fact, it is defined to be as one less than the size of the largest clique, to ensure that a tree has a treewidth of 1.

Because there in general there can be multiple different junction trees that can be constructed from a given starting graph, the treewidth is defined by the junction tree for which the largest clique has the fewest variables.

If the treewidth of the original graph is high, the junction tree algorithm becomes impractical.

Starting with a DAG
The Junction Tree Algorithm: Moralizing

- Moralize by marrying the parents of each node
- Then remove edge directions
- This results in an undirected graph with no additional conditional independence relations
- Triangulate so that there is no loop of length > 3 (before we had loop BCED) without a chord (not unique step)
- This is necessary so that the final junction tree satisfies the "running intersection property"
Find cliques of the moralized, triangulated graph
Junction Tree Algorithm

Start with a DAG

Moralization (marrying the parents)

Triangulate so that there is no loop of length > 3 without a chord. This is necessary so that the final junction tree satisfies the running intersection property

Find all the cliques
Junction Tree Algorithm

- Form junction tree: tree of (overlapping) sets of variables
- The running intersection property means that if a variable appears in more than one clique (e.g. C), it appears in all intermediate cliques in the tree.
- The junction tree propagation algorithm ensures that neighboring cliques have consistent probability distribution
- *Local consistency → global consistency*
Inference in Graphs With Loops

- For many problems of practical interest, we need to exploit approximation methods.
- An important class of such approximations are variational methods.
- Complementing these deterministic approaches is a wide range of sampling (Monte Carlo) methods.
- Here we consider approximate inference in graphs with loops, building directly on the exact inference in trees.
- The idea is simply to apply the sum-product algorithm even though there is no guarantee that it will yield good results. This approach is known as loopy belief propagation and is possible because the message passing rules for the sum-product algorithm are purely local.

Loopy Belief Propagation

- Sum-Product on general graphs.

- Initial unit messages passed across all links, after which messages are passed around until convergence (not guaranteed!).

- Approximate but tractable for large graphs.

- Sometime works well, sometimes not at all.
Loopy Belief Propagation

- Because the graph now has cycles, information can flow many times around the graph. For some models the algorithm will converge, whereas for others it will not.

- We need to define a message passing schedule. Let us assume that one message is passed at a time on any given link and in any given direction. Each message sent from a node replaces any previous message sent in the same direction across the same link and will itself be a function only of the most recent messages received by that node at previous steps of the algorithm.

- We have seen that a message can only be sent across a link from a node when all other messages have been received by that node across its other links.

- Because there are loops in the graph, this raises the problem of how to initiate the message passing algorithm.
Loopy Belief Propagation

- Suppose that an initial message given by the unit function has been passed across every link in each direction. Every node is then in a position to send a message.

- There are now many possible ways to organize the message passing schedule. The *flooding schedule* simultaneously passes a message across every link in both directions at each time step, whereas schedules that pass one message at a time are called *serial schedules*.

- A (variable or factor) node \( a \) has a *message pending* on its link to a node \( b \) if node \( a \) has received any message on any of its other links since the last time it send a message to \( b \).

- Thus, when a node receives a message on one of its links, this creates pending messages on all of its other links. Only pending messages need to be transmitted because other messages would simply duplicate the previous message on the same link.

For graphs with a tree structure, any schedule that sends only pending messages will eventually terminate once a message has passed in each direction across every link (can be proved by induction).

- At this point, there are no pending messages, and the product of the received messages at every variable gives the exact marginal.

In graphs with loops, however, the algorithm may never terminate because there might always be pending messages. In practice it is found to converge within a reasonable time.

- If a graph has one or more cycles, there will always be at least one pending message regardless how long the algorithm runs (when a node sends a pending message to the next node on the cycle, that generates a pending message on the next edge on the cycle).

Once the algorithm has converged, or once it has been stopped if convergence is not observed, the (approximate) local marginals can be computed using the product of the most recently received incoming messages to each variable node or factor node on every link.
In some applications, the loopy belief propagation algorithm can give poor results, whereas in other applications it has proven to be very effective.

In particular, state-of-the-art algorithms for decoding certain kinds of error-correcting codes are equivalent to loopy belief propagation.

Learning the Graph Structure

- Up to now, we have assumed that the structure of the graph is known. Here, we want to learn the graph structure from data.
- Need to define a space of possible structures and a measure to score each structure.
- If we have a prior $p(m)$ over graphs indexed by $m$, then the posterior distribution is given by
  \[ p(m|\mathcal{D}) \propto p(m)p(\mathcal{D}|m) \]
  where $\mathcal{D}$ is the observed data set.
- The model evidence $p(\mathcal{D}|m)$ then provides the score for each model. Evaluation of $p(\mathcal{D}|m)$ involves marginalization over the latent variables which is a a challenging computational problem.
- Because the number of different graph structures grows exponentially with the number of nodes, it is often necessary to resort to heuristics to find good candidates.