Announcements

- Exercise 5 now on the web
  - Junction tree algorithm (today)
  - Markov random fields
  - Graph Cuts (next lecture)
  - Due date: Thursday in TWO weeks (14.07.)

- Next week
  - Exercise 4 on Tuesday
  - Bayes Nets
  - Conditional Independence, D-Separation, Bayes Ball
  - Factor graphs (today)
  - Sum-Product algorithm (today)
  ⇒ Submit your solutions by Monday evening!
Course Outline

• Fundamentals (2 weeks)
  - Bayes Decision Theory
  - Probability Density Estimation

• Discriminative Approaches (5 weeks)
  - Linear Discriminant Functions
  - Statistical Learning Theory & SVMs
  - Ensemble Methods & Boosting
  - Decision Trees & Randomized Trees

• Generative Models (4 weeks)
  - Bayesian Networks
  - Markov Random Fields
  - Exact Inference
Recap: Undirected Graphical Models

- Undirected graphical models ("Markov Random Fields")
  - Given by undirected graph

- Conditional independence for undirected graphs
  - If every path from any node in set $A$ to set $B$ passes through at least one node in set $C$, then $A \perp B|C$.
  - Simple Markov blanket:
Recap: Factorization in MRFs

- Joint distribution
  - Written as product of potential functions over maximal cliques in the graph:
  \[ p(x) = \frac{1}{Z} \prod_C \psi_C(x_C) \]
  - The normalization constant \( Z \) is called the partition function.
  \[ Z = \sum_x \prod_C \psi_C(x_C) \]

- Remarks
  - BNs are automatically normalized. But for MRFs, we have to explicitly perform the normalization.
  - Presence of normalization constant is major limitation!
    - Evaluation of \( Z \) involves summing over \( \mathcal{O}(K^M) \) terms for \( M \) nodes!
Recap: Factorization in MRFs

- **Role of the potential functions**
  - General interpretation
    - No restriction to potential functions that have a specific probabilistic interpretation as marginals or conditional distributions.
  - Convenient to express them as exponential functions ("Boltzmann distribution")
    \[ \psi_C(x_C) = \exp\{-E(x_C)\} \]
    - with an energy function \( E \).
  - Why is this convenient?
    - Joint distribution is the product of potentials \( \Rightarrow \) sum of energies.
    - We can take the log and simply work with the sums...
Recap: Converting Directed to Undirected Graphs

- Problematic case: multiple parents

\[ p(x) = p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3) \]

Need a clique of \( x_1, ..., x_4 \) to represent this factor!

- Need to introduce additional links ("marry the parents").
  \[ \Rightarrow \] This process is called **moralization**. It results in the moral graph.

Slide adapted from Chris Bishop

B. Leibe

Image source: C. Bishop, 2006
Recap: Conversion Algorithm

- General procedure to convert directed → undirected
  1. Add undirected links to marry the parents of each node.
  2. Drop the arrows on the original links ⇒ moral graph.
  3. Find maximal cliques for each node and initialize all clique potentials to 1.
  4. Take each conditional distribution factor of the original directed graph and multiply it into one clique potential.

- Restriction
  - Conditional independence properties are often lost!
  - Moralization results in additional connections and larger cliques.
Recap: Computing Marginals

• How do we apply graphical models?
  - Given some observed variables, we want to compute distributions of the unobserved variables.
  - In particular, we want to compute marginal distributions, for example $p(x_4)$.

• How can we compute marginals?
  - Classical technique: *sum-product algorithm* by Judea Pearl.
  - In the context of (loopy) undirected models, this is also called (loopy) *belief propagation* [Weiss, 1997].
  - Basic idea: message-passing.
Recap: Message Passing on a Chain

- **Idea**
  - Pass messages from the two ends towards the query node $x_n$.

- **Define the messages recursively:**
  $\mu_\alpha(x_n) = \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \mu_\alpha(x_{n-1})$
  $\mu_\beta(x_n) = \sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1}) \mu_\beta(x_{n+1})$

- **Compute the normalization constant $Z$ at any node $x_m$.**
  $Z = \sum_{x_n} \mu_\alpha(x_n) \mu_\beta(x_n)$
Summary: Inference on a Chain

- To compute local marginals:
  - Compute and store all forward messages $\mu_\alpha(x_n)$.
  - Compute and store all backward messages $\mu_\beta(x_n)$.
  - Compute $Z$ at any node $x_m$.
  - Compute
    \[
    p(x_n) = \frac{1}{Z} \mu_\alpha(x_n) \mu_\beta(x_n)
    \]
    for all variables required.

- Inference through message passing
  - We have thus seen a first message passing algorithm.
  - How can we generalize this?
Inference on Trees

- Let’s next assume a tree graph.
  - Example:

  - We are given the following joint distribution:
    \[ p(A, B, C, D, E) = \frac{1}{Z} f_1(A, B) \cdot f_2(B, D) \cdot f_3(C, D) \cdot f_4(D, E) \]

  - Assume we want to know the marginal \( p(E) \)...
Inference on Trees

- **Strategy**
  - Marginalize out all other variables by summing over them.

- Then rearrange terms:

\[
p(E) = \sum_A \sum_B \sum_C \sum_D p(A, B, C, D, E)
\]

\[
= \sum_A \sum_B \sum_C \sum_D \frac{1}{Z} f_1(A, B) \cdot f_2(B, D) \cdot f_3(C, D) \cdot f_4(D, E)
\]

\[
= \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot \left( \sum_B f_2(B, D) \cdot \left( \sum_A f_1(A, B) \right) \right) \right)
\]
**Marginalization with Messages**

- **Use messages** to express the marginalization:

  \[
  m_{A \rightarrow B} = \sum_A f_1(A, B) \quad m_{C \rightarrow D} = \sum_C f_3(C, D)
  \]

  \[
  m_{B \rightarrow D} = \sum_B f_2(B, D)m_{A \rightarrow B}(B)
  \]

  \[
  m_{D \rightarrow E} = \sum_D f_4(D, E)m_{B \rightarrow D}(D)m_{C \rightarrow D}(D)
  \]

  \[
p(E) = \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot \left( \sum_B f_2(B, D) \cdot \left( \sum_A f_1(A, B) \right) \right) \right)
  \]

  \[
  = \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot \left( \sum_B f_2(B, D) \cdot m_{A \rightarrow B}(B) \right) \right)
  \]
Marginalization with Messages

- **Use messages** to express the marginalization:

\[
\begin{align*}
m_{A \rightarrow B} &= \sum_A f_1(A, B) \\
m_{C \rightarrow D} &= \sum_C f_3(C, D) \\
m_{B \rightarrow D} &= \sum_B f_2(B, D)m_{A \rightarrow B}(B) \\
m_{D \rightarrow E} &= \sum_D f_4(D, E)m_{B \rightarrow D}(D)m_{C \rightarrow D}(D) \\
p(E) &= \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot \left( \sum_B f_2(B, D) \cdot \left( \sum_A f_1(A, B) \right) \right) \right) \\
&= \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot m_{B \rightarrow D}(D) \right)
\end{align*}
\]
Marginalization with Messages

- **Use messages** to express the marginalization:

  \[ m_{A\rightarrow B} = \sum_{A} f_1(A, B) \quad m_{C\rightarrow D} = \sum_{C} f_3(C, D) \]

  \[ m_{B\rightarrow D} = \sum_{B} f_2(B, D)m_{A\rightarrow B}(B) \]

  \[ m_{D\rightarrow E} = \sum_{D} f_4(D, E)m_{B\rightarrow D}(D)m_{C\rightarrow D}(D) \]

\[
p(E) = \frac{1}{Z} \left( \sum_{D} f_4(D, E) \cdot \left( \sum_{C} f_3(C, D) \right) \cdot \left( \sum_{B} f_2(B, D) \cdot \left( \sum_{A} f_1(A, B) \right) \right) \right) \]

\[
= \frac{1}{Z} \left( \sum_{D} f_4(D, E) \cdot m_{C\rightarrow D}(D) \cdot m_{B\rightarrow D}(D) \right)
\]

---

Slide credit: Bernt Schiele, Stefan Roth
Marginalization with Messages

- **Use messages** to express the marginalization:

  
  \[
  m_{A\rightarrow B} = \sum_A f_1(A, B) \quad m_{C\rightarrow D} = \sum_C f_3(C, D)
  \]

  
  \[
  m_{B\rightarrow D} = \sum_B f_2(B, D)m_{A\rightarrow B}(B) \quad m_{D\rightarrow E} = \sum_D f_4(D, E)m_{B\rightarrow D}(D)m_{C\rightarrow D}(D)
  \]

  
  \[
  p(E) = \frac{1}{Z} \left( \sum_D f_4(D, E) \cdot \left( \sum_C f_3(C, D) \right) \cdot \left( \sum_B f_2(B, D) \cdot \left( \sum_A f_1(A, B) \right) \right) \right)
  \]

  
  \[
  = \frac{1}{Z} m_{D\rightarrow E}(E)
  \]
Recap: Message Passing on Trees

- **General procedure** for all tree graphs.
  - Root the tree at the variable that we want to compute the marginal of.
  - Start computing messages at the leaves.
  - Compute the messages for all nodes for which all incoming messages have already been computed.
  - Repeat until we reach the root.

- If we want to compute the marginals for all possible nodes (roots), we can reuse some of the messages.
  - Computational expense linear in the number of nodes.

- We already motivated message passing for inference.
  - How can we formalize this into a general algorithm?
How Can We Generalize This?

- Message passing algorithm motivated for trees.
  - Now: generalize this to directed polytrees.
  - We do this by introducing a common representation
    ⇒ Factor graphs
Topics of This Lecture

- **Factor graphs**
  - Construction
  - Properties

- **Sum-Product Algorithm for computing marginals**
  - Key ideas
  - Derivation
  - Example

- **Max-Sum Algorithm for finding most probable value**
  - Key ideas
  - Derivation
  - Example

- **Algorithms for loopy graphs**
  - Junction Tree algorithm
  - Loopy Belief Propagation
Factor Graphs

**Motivation**

- Joint probabilities on both directed and undirected graphs can be expressed as a product of factors over subsets of variables.
- **Factor graphs** make this decomposition explicit by introducing separate nodes for the factors.

\[
p(x) = \frac{1}{Z} f_a(x_1, x_2) f_b(x_1, x_2) f_c(x_2, x_3) f_d(x_3)
\]

\[
= \frac{1}{Z} \prod_s f_s(x_s)
\]

Image source: C. Bishop, 2006

Slide adapted from Chris Bishop
Factor Graphs from Directed Graphs

Conversion procedure
1. Take variable nodes from directed graph.
2. Create factor nodes corresponding to conditional distributions.
3. Add the appropriate links.

⇒ Different factor graphs possible for same directed graph.

\[ p(x) = p(x_1)p(x_2) \]
\[ f(x_1, x_2, x_3) = \frac{p(x_3|x_1, x_2)}{\frac{p(x_1)p(x_2)p(x_3|x_1, x_2)}{p(x_1)p(x_2)p(x_3|x_1, x_2)}} \]

\[ f_a(x_1) = p(x_1) \]
\[ f_b(x_2) = p(x_2) \]
\[ f_c(x_1, x_2, x_3) = p(x_3|x_1, x_2) \]
Factor Graphs from Undirected Graphs

- Some factor graphs for the same undirected graph:

$\psi(x_1, x_2, x_3)$

$f(x_1, x_2, x_3) = \psi(x_1, x_2, x_3)$

$f_a(x_1, x_2, x_3) f_b(x_2, x_3) = \psi(x_1, x_2, x_3)$

$\Rightarrow$ The factor graph keeps the factors explicit and can thus convey more detailed information about the underlying factorization!
Factor Graphs - Why Are They Needed?

- Converting a directed or undirected tree to factor graph
  - The result will again be a tree.

- Converting a directed polytree
  - Conversion to undirected tree creates loops due to moralization!
  - Conversion to a factor graph again results in a tree.
Topics of This Lecture

- Factor graphs
  - Construction
  - Properties

- Sum-Product Algorithm for computing marginals
  - Key ideas
  - Derivation
  - Example

- Max-Sum Algorithm for finding most probable value
  - Key ideas
  - Derivation
  - Example

- Algorithms for loopy graphs
  - Junction Tree algorithm
  - Loopy Belief Propagation
Sum-Product Algorithm

- **Objectives**
  - Efficient, exact inference algorithm for finding marginals.
  - In situations where several marginals are required, allow computations to be shared efficiently.

- **General form of message-passing idea**
  - Applicable to tree-structured factor graphs.
    ⇒ Original graph can be undirected tree or directed tree/polytree.

- **Key idea: Distributive Law**
  \[ ab + ac = a(b + c) \]
  ⇒ Exchange summations and products exploiting the tree structure of the factor graph.
  - Let’s assume first that all nodes are hidden (no observations).

Slide adapted from Chris Bishop
**Sum-Product Algorithm**

- **Goal:**
  - Compute marginal for $x$: $p(x) = \sum_{x \setminus x} p(x)$
  - Tree structure of graph allows us to partition the joint distrib. into groups associated with each neighboring factor node:
    $$p(x) = \prod_{s \in \text{ne}(x)} F_s(x, X_s)$$

---

Image source: C. Bishop, 2006
Sum-Product Algorithm

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

- Exchanging products and sums:
  \[ p(x) = \prod_{s \in \text{ne}(x)} \left[ \sum_{X_s} F_s(x, X_s) \right] = \prod_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x) \]

\text{ne}(x): \text{neighbors of } x.

Image source: C. Bishop, 2006

Slide adapted from Chris Bishop
Sum-Product Algorithm

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

- Exchanging products and sums:
  \[ p(x) = \prod_{s \in \text{ne}(x)} \left[ \sum_{X_s} F_s(x, X_s) \right] = \prod_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x) \]

This defines a first type of message \( \mu_{f_s \rightarrow x}(x) \):

\[ \mu_{f_s \rightarrow x}(x) \equiv \sum_{X_s} F_s(x, X_s) \]
Sum-Product Algorithm

First message type:

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

- Evaluating the messages:
  - Each factor \( F_s(x, X_s) \) is again described by a factor (sub-)graph.
  - Can itself be factorized:

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

First message type:
\[ \mu_{f_s \rightarrow x}(x) \equiv \sum_{X_s} F_s(x, X_s) \]

- Evaluating the messages:
  - Thus, we can write

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

First message type:
\[
\mu_{f_s\rightarrow x}(x) \equiv \sum_{X_s} F_s(x, X_s)
\]

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

- Evaluating the messages:
  - Thus, we can write
  
  \[
  \mu_{f_s\rightarrow x}(x) = \sum_{x_1} \ldots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m \in \text{ne}(f_s) \setminus x} \left[ \sum_{X_{sm}} G_m(x_m, X_{sm}) \right] 
  \]
  
  \[
  = \sum_{x_1} \ldots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m\rightarrow f_s}(x_m)
  \]

B. Leibe

Image source: C. Bishop, 2006
Sum-Product Algorithm

Recursive definition:

\[ \mu_{f_l \rightarrow x_m}(x_m) \equiv \sum_{X_{sm}} F_l(x_m, X_{sm}) \]

Each term \( G_m(x_m, X_{sm}) \) is again given by a product

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

• Recursive message evaluation:
  - Exchanging sum and product, we again get

\[
\mu_{x_m \rightarrow f_s}(x_m) \equiv \sum_{X_{sm}} G_m(x_m, X_{sm}) = \sum_{X_{sm}} \prod_{l \in \text{ne}(x_m) \setminus f_s} F_l(x_m, X_{ml}) = \prod_{l \in \text{ne}(x_m) \setminus f_s} \mu_{f_l \rightarrow x_m}(x_m)
\]

B. Leibe

Image source: C. Bishop, 2006
Sum-Product Algorithm - Summary

- Two kinds of messages
  - **Message from factor node to variable nodes:**
    - **Sum** of factor contributions
      \[ \mu_{f_s \rightarrow x}(x) \equiv \sum_{X_s} F_s(x, X_s) \]
      \[ = \sum_{X_s} f_s(x_s) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \]
  - **Product** of incoming messages
    \[ \mu_{x_m \rightarrow f_s}(x_m) \equiv \prod_{l \in \text{ne}(x_m) \setminus f_s} \mu_{f_l \rightarrow x_m}(x_m) \]

  ⇒ Simple propagation scheme.
Sum-Product Algorithm

- **Initialization**
  - Start the recursion by sending out messages from the leaf nodes

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

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

- **Propagation procedure**
  - A node can send out a message once it has received incoming messages from all other neighboring nodes.
  - Once a variable node has received all messages from its neighboring factor nodes, we can compute its marginal by multiplying all messages and renormalizing:

\[
p(x) \propto \prod_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x)
\]

Image source: C. Bishop, 2006
Sum-Product Algorithm - Summary

• To compute local marginals:
  - Pick an arbitrary node as 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.

• Computational effort
  - Total number of messages = 2 \cdot \text{number of links in the graph.}
  - Maximal parallel runtime = 2 \cdot \text{tree height.}
Sum-Product: Example

Unnormalized joint distribution:
\[ \tilde{p}(\mathbf{x}) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4) \]

We want to compute the values of all marginals...

Picking \( x_3 \) as root...
\[ \Rightarrow x_1 \text{ and } x_4 \text{ are leaves.} \]
Sum-Product: Example

Message definitions:

\[ \mu_{f_s \rightarrow x}(x) \equiv \sum_{x_s} f_s(x_s) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \]

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

\[
\begin{align*}
\mu_{x_1 \rightarrow f_a}(x_1) &= 1 \\
\mu_{f_a \rightarrow x_2}(x_2) &= \sum_{x_1} f_a(x_1, x_2) \\
\mu_{x_4 \rightarrow f_c}(x_4) &= 1 \\
\mu_{f_c \rightarrow x_2}(x_2) &= \sum_{x_4} f_c(x_2, x_4) \\
\mu_{x_2 \rightarrow f_b}(x_2) &= \mu_{f_a \rightarrow x_2}(x_2) \mu_{f_c \rightarrow x_2}(x_2)
\end{align*}
\]
Perceptual and Sensory Augmented Computing

Sum-Product: Example

Message definitions:

\[ \mu_{f_s \rightarrow x}(x) = \sum_{x_s} f_s(x_s) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \]

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

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

Message definitions:

\[ \mu_{f_s \to x}(x) \equiv \sum_{x_s} f_s(x_s) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \to f_s}(x_m) \]

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

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

Message definitions:

\[ \mu_{f_s \rightarrow x}(x) \equiv \sum_{x_s} f_s(x_s) \prod_{m \in \text{ne}(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \]

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

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

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

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

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

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

\[ \mu_{f_c \rightarrow x_4}(x_4) = \sum_{x_2} f_c(x_2, x_4) \mu_{x_2 \rightarrow f_c}(x_2) \]
Sum-Product: Example

Verify that marginal is correct:

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

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

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

\[
= \sum_{x_1} \sum_{x_3} \sum_{x_4} \tilde{p}(x)
\]
Sum-Product Algorithm - Extensions

- **Dealing with observed nodes**
  - Until now we had assumed that all nodes were hidden...
  - Observed nodes can easily be incorporated:
    - Partition $x$ into hidden variables $h$ and observed variables $v = \hat{v}$.
    - Simply multiply the joint distribution $p(x)$ by
      \[
      \prod_i I(v_i, \hat{v}_i)
      \]
      where
      \[
      I(v_i, \hat{v}_i) = \begin{cases} 
      1, & \text{if } v_i = \hat{v}_i \\ 
      0, & \text{else.} 
      \end{cases}
      \]
      ⇒ Any summation over variables in $v$ collapses into a single term.

- **Further generalizations**
  - So far, assumption that we are dealing with discrete variables.
  - But the sum-product algorithm can also be generalized to simple continuous variable distributions, e.g. linear-Gaussian variables.
Topics of This Lecture

- Factor graphs
  - Construction
  - Properties
- Sum-Product Algorithm for computing marginals
  - Key ideas
  - Derivation
  - Example
- Max-Sum Algorithm for finding most probable value
  - Key ideas
  - Derivation
  - Example
- Algorithms for loopy graphs
  - Junction Tree algorithm
  - Loopy Belief Propagation
Max-Sum Algorithm

• Objective: an efficient algorithm for finding
  - Value \( x^{\text{max}} \) that maximises \( p(x) \);
  - Value of \( p(x^{\text{max}}) \).
  \[ \Rightarrow \text{Application of dynamic programming in graphical models.} \]

• In general, maximum marginals \( \neq \) joint maximum.
  - Example:
    \[
    \begin{array}{c|cc}
      & x = 0 & x = 1 \\
      \hline
      y = 0 & 0.3 & 0.4 \\
      y = 1 & 0.3 & 0.0 \\
    \end{array}
    \]
    \[
    \arg \max_x p(x, y) = 1 \quad \arg \max_x p(x) = 0
    \]
Max-Sum Algorithm - Key Ideas

• Key idea 1: Distributive Law (again)

\[
\max(ab, ac) = a \max(b, c)
\]
\[
\max(a + b, a + c) = a + \max(b, c)
\]

⇒ Exchange products/summations and max operations exploiting the tree structure of the factor graph.

• Key idea 2: Max-Product → Max-Sum

⇒ We are interested in the maximum value of the joint distribution

\[
p(x^{\text{max}}) = \max_{x} p(x)
\]

⇒ Maximize the product \(p(x)\).

⇒ For numerical reasons, use the logarithm.

\[
\ln \left( \max_{x} p(x) \right) = \max_{x} \ln p(x).
\]

⇒ Maximize the sum (of log-probabilities).

B. Leibe
Max-Sum Algorithm

- Maximizing over a chain (max-product)

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

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

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

- Exchange max and product operators

- Generalizes to tree-structured factor graph

\[
\max_x p(x) = \max_{x_n} \prod_{f_s \in \text{ne}(x_n)} \max_{X_s} f_s(x_n, X_s)
\]
Max-Sum Algorithm

• Initialization (leaf nodes)

\[ \mu_{x \rightarrow f}(x) = 0 \quad \mu_{f \rightarrow x}(x) = \ln f(x) \]

• Recursion

  ➢ Messages

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

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

  ➢ For each node, keep a record of which values of the variables gave rise to the maximum state:

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

- Termination (root node)
  - Score of maximal configuration
    \[ p^{\max} = \max_x \left[ \sum_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x) \right] \]
  - Value of root node variable giving rise to that maximum
    \[ x^{\max} = \arg\max_x \left[ \sum_{s \in \text{ne}(x)} \mu_{f_s \rightarrow x}(x) \right] \]
  - Back-track to get the remaining variable values
    \[ x_{n-1}^{\max} = \phi(x_n^{\max}) \]
Visualization of the Back-Tracking Procedure

- Example: Markov chain

⇒ Same idea as in Viterbi algorithm for HMMs...

Slide adapted from Chris Bishop
Topics of This Lecture

- Factor graphs
  - Construction
  - Properties

- Sum-Product Algorithm for computing marginals
  - Key ideas
  - Derivation
  - Example

- Max-Sum Algorithm for finding most probable value
  - Key ideas
  - Derivation
  - Example

- Algorithms for loopy graphs
  - Junction Tree algorithm
  - Loopy Belief Propagation
Junction Tree Algorithm

• Motivation
  - Exact inference on general graphs.
  - Works by turning the initial graph into a junction tree and then running a sum-product-like algorithm.
  - Intractable on graphs with large cliques.

• Main steps
  1. If starting from directed graph, first convert it to an undirected graph by moralization.
  2. Introduce additional links by triangulation in order to reduce the size of cycles.
  3. Find cliques of the moralized, triangulated graph.
  4. Construct a new graph from the maximal cliques.
  5. Remove minimal links to break cycles and get a junction tree.
     ⇒ Apply regular message passing to perform inference.
Junction Tree Algorithm

- Starting from an undirected graph...
1. Convert to an undirected graph through **moralization**.
   - Marry the parents of each node.
   - Remove edge directions.
2. Triangulate

- Such 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 (explained later).
Junction Tree Algorithm

3. Find cliques of the moralized, triangulated graph.
4. Construct a new **junction graph** from maximal cliques.
   - Create a node from each clique.
   - Each link carries a list of all variables in the intersection.
     - Drawn in a “**separator**” box.

B. Leibe

Image source: Z. Gharahmani
5. Remove links to break cycles ⇒ junction tree.
   - For each cycle, remove the link(s) with the minimal number of shared nodes until all cycles are broken.
   - Result is a maximal spanning tree, the junction tree.
• **Running intersection property**
  
  - "If a variable appears in more than one clique, it also appears in all intermediate cliques in the tree".
  - This ensures that neighboring cliques have consistent probability distributions.
  - Local consistency → global consistency
Junction Tree: Example 1

- Algorithm
  1. Moralization
  2. Triangulation (not necessary here)
Junction Tree: Example 1

- Algorithm
  1. Moralization
  2. Triangulation (not necessary here)
  3. Find cliques
  4. Construct junction graph

(b) Moral graph

(c) Junction graph

Image source: J. Pearl, 1988
Junction Tree: Example 1

- **Algorithm**
  1. Moralization
  2. Triangulation (not necessary here)
  3. Find cliques
  4. Construct junction graph
  5. Break links to get junction tree
Junction Tree: Example 2

- Without triangulation step
  - The final graph will contain cycles that we cannot break without losing the running intersection property!

Image source: J. Pearl, 1988
• When applying the triangulation
  - Only small cycles remain that are easy to break.
  - Running intersection property is maintained.
Junction Tree Algorithm

• Good news
  - The junction tree algorithm is efficient in the sense that for a given graph there does not exist a computationally cheaper approach.

• Bad news
  - This may still be too costly.
  - Effort determined by number of variables in the largest clique.
  - Grows exponentially with this number (for discrete variables).
  ⇒ Algorithm becomes impractical if the graph contains large cliques!
Loopy Belief Propagation

- Alternative algorithm for loopy graphs
  - Sum-Product on general graphs.
  - Strategy: simply ignore the problem.
  - Initial unit messages passed across all links, after which messages are passed around until convergence
    - Convergence is not guaranteed!
    - Typically break off after fixed number of iterations.
  - Approximate but tractable for large graphs.
  - Sometime works well, sometimes not at all.
References and Further Reading

• A thorough introduction to Graphical Models in general and Bayesian Networks in particular can be found in Chapter 8 of Bishop’s book.

Christopher M. Bishop
Pattern Recognition and Machine Learning
Springer, 2006