Recap: Conditional Independence

- Three cases
  - Divergent ("Tail-to-Tail")
    - Conditional independence when c is observed.
  - Chain ("Head-to-Tail")
    - Conditional independence when c is observed.
  - Convergent ("Head-to-Head")
    - Conditional independence when neither c, nor any of its descendants are observed.

Recap: "Bayes Ball" Algorithm

- Graph algorithm to compute d-separation
  - Goal: Get a ball from X to Y without being blocked by Z.
  - Depending on its direction and the previous node, the ball can:
    - Pass through (from parent to all children, from child to all parents)
    - Bounce back (from any parent/child to all parents/children)
    - Be blocked

- Game rules
  - An unobserved node (W ∈ V) passes through balls from parents, but also bounces back balls from children.
  - An observed node (W ∈ V) bounces back balls from parents, but blocks balls from children.

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 ⊥⊥ B given C.
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.

- 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 \( O(K^M) \) terms for \( M \) nodes!

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

  \[ x_1 \rightarrow x_2 \rightarrow x_3 \]

  Need to introduce additional links ("marry the parents").

  This process is called moralization. It results in the moral graph.

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 \( p(x_j) \).

- 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 \( x_q \).

- Define the messages recursively:
  \[ \mu_a(x_a) = \sum_{x_{a-1}} \psi_{a-1,a}(x_{a-1}, x_a) \mu_{a-1}(x_{a-1}) \]
  \[ \mu_b(x_a) = \sum_{x_{a+1}} \psi_{a,a+1}(x_a, x_{a+1}) \mu_{a+1}(x_{a+1}) \]

- Compute the normalization constant \( Z \) at any node \( x_q \):
  \[ Z = \sum_{x_q} \mu_a(x_a) \mu_b(x_a) \]
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.

  - Joint probability
    \[ p(x) = \frac{1}{Z} f_1(x_1, x_2) f_2(x_1, x_2) f_3(x_2, x_3) f_4(x_3) \]
    \[ Z = \prod f_i(x_i) \n
Factor Graphs from Directed Graphs

- Conversion procedure
  - \[ f_1(x_1, x_2, x_3) = f_2(x_1, x_2) \]
  - \[ f_3(x, x_1) = f_3(x, x_2) \]
  - \[ f_4(x_3) = f_4(x_3) \]

  - Different factor graphs possible for same directed graph.

Factor Graphs from Undirected Graphs

- Some factor graphs for the same undirected graph:
  - \[ f_1(x_1, x_2, x_3) \]
  - \[ f_2(x_1, x_2) \]
  - \[ f_3(x_2, x_3) \]
  - \[ f_4(x_3) \]

  - 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
  - Exchange summations and products exploiting the tree structure of the factor graph.
  - Let’s assume first that all nodes are hidden (no observations).

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

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

- **First message type:**
  \[ \mu_{s\rightarrow m}(x) = \sum_{x_i} F_i(x, X_i) \]

- **Second message type:**
  \[ \mu_{m\rightarrow s}(x_m) = \sum_{x_m} G_m(x_m, X_m) \]

- **Evaluating the messages:**
  - Each factor \( F_i(x, X_i) \) is again described by a factor (sub-)graph.
  - Can itself be factorized:
    \[ F_i(x, X_i) = f_i(x, x_1, \ldots, x_M)G_1(x_1, X_1) \ldots G_M(x_M, X_M) \]

- **Recursive message evaluation:**
  - Thus, we can write
    \[ \mu_{s\rightarrow m}(x) = \sum_{x_i} \sum_{x_m} f_i(x, x_1, \ldots, x_M) \prod_{m \notin \text{in}(f_i[x])} G_m(x_m, X_m) \]
    \[ = \sum_{x_i} \sum_{x_m} f_i(x, x_1, \ldots, x_M) \prod_{m \notin \text{in}(f_i[x])} \mu_{m\rightarrow s}(x_m) \]
  - Exchanging sum and product, we again get
    \[ \mu_{s\rightarrow m}(x) = \sum_{x_m} G_m(x_m, X_m) \sum_{x_i} \prod_{m \notin \text{in}(f_i[x])} \mu_{m\rightarrow s}(x_m) \]
    \[ = \sum_{x_m} G_m(x_m, X_m) \prod_{m \notin \text{in}(f_i[x])} \mu_{m\rightarrow s}(x_m) \]

- **Two kinds of messages**
  - Message from factor node to variable nodes:
    - Sum of factor contributions
      \[ \mu_{f\rightarrow s}(x) = \sum_{X_i} F_i(x, X_i) \]
      \[ = \sum_{X_i} f_i(x_i) \prod_{m \notin \text{in}(f_i[x])} \mu_{m\rightarrow s}(x_m) \]
  - Message from variable node to factor node:
    - Product of incoming messages
      \[ \mu_{s\rightarrow f}(x_m) = \prod_{l \in \text{in}(f_l[x]), l \notin f_i} \mu_{l\rightarrow s}(x_m) \]

- **Simple propagation scheme.**
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 · number of links in the graph.
  - Maximal parallel runtime = 2 · tree height.

Unnormalized joint distribution:
$$p(x) = \prod_{i=1}^{n} p(x_i)$$

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

Message definitions:
- Sum-Product: Example

$$\sum_{x_i} \prod_{j \neq i} \mu_{x_j \rightarrow x_i}(x_j) \rightarrow x_i$$

$$\mu_{x_i \rightarrow x_j}(x_i) \equiv \prod_{k \neq i} \mu_{x_k \rightarrow x_i}(x_k) x_j$$
**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 $v = \tilde{v}$.
    - Simply multiply the joint distribution $p(x)$ by
      $$
      \prod_{i} I(v_i, \hat{v}_i) \text{ where } I(v_i, \hat{v}_i) = \begin{cases} 1, & \text{if } v_i = \hat{v}_i \\ 0, & \text{else.} \end{cases}
      $$
    - 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.

**Sum-Product Algorithm - Key Ideas**

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

- In general, maximum marginals are joint maximum.
  - Example:
    $$
    \begin{array}{c|ccc}
    \gamma & y = 0 & y = 1 \\
    \delta & 0.2 & 0.4 \\
    \epsilon & 0.8 & 0.2 \\
    \end{array}
    $$
    $$
    \arg\max_{\delta} p(x, y) = 1. \quad \arg\max_{\epsilon} p(x) = 0
    $$
Max-Sum Algorithm

- Initialization (leaf nodes)
  \[ p_{\text{init}}(x) = \prod p_f(x) \]

- Recursion
  - Messages
    \[ p_{\text{rec}}(x) = \text{maximize } \left( \log f(x_1, x_2, \ldots, x_k) + \sum_{\text{other variables}} p_{\text{rec}}(x_0) \right) \]
    \[ \theta_{\text{rec}}(x) = \sum_{\text{other variables}} p_{\text{rec}}(x_0) \]
  - For each node, keep a record of which values of the variables gave rise to the maximum state:
    \[ \phi(x) = \underbrace{\log f(x_1, x_2, \ldots, x_k) + \sum_{\text{other variables}} p_{\text{rec}}(x_0)}_{\text{maximal}} \]

- Termination (root node)
  - Score of maximal configuration
    \[ \phi_{\text{max}} = \max_{x} \left( \sum_{\text{other variables}} p_f(x) \right) \]
  - Value of root node variable giving rise to that maximum
  - Back-track to get the remaining variable values
    \[ x_{\text{final}} = \phi_{\text{max}} \]

Visualization of the Back-Tracking Procedure

- Example: Markov chain

Topics of This Lecture

- Factor graphs
  - Construction
  - Properties
- Sum-Product Algorithm for computing marginals
  - Key Idea
  - Derivation
  - Example
- Max-Sum Algorithm for finding most probable value
  - Key Idea
  - 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.

Junction Tree Algorithm

- Starting from an undirected graph...
Junction Tree Algorithm

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).

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.

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.

Junction Tree – Properties

- 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 2**

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

**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