Computer Science

CSC535: Probabilistic Graphical Models

Message Passing Inference

Prof. Jason Pacheco

Outline

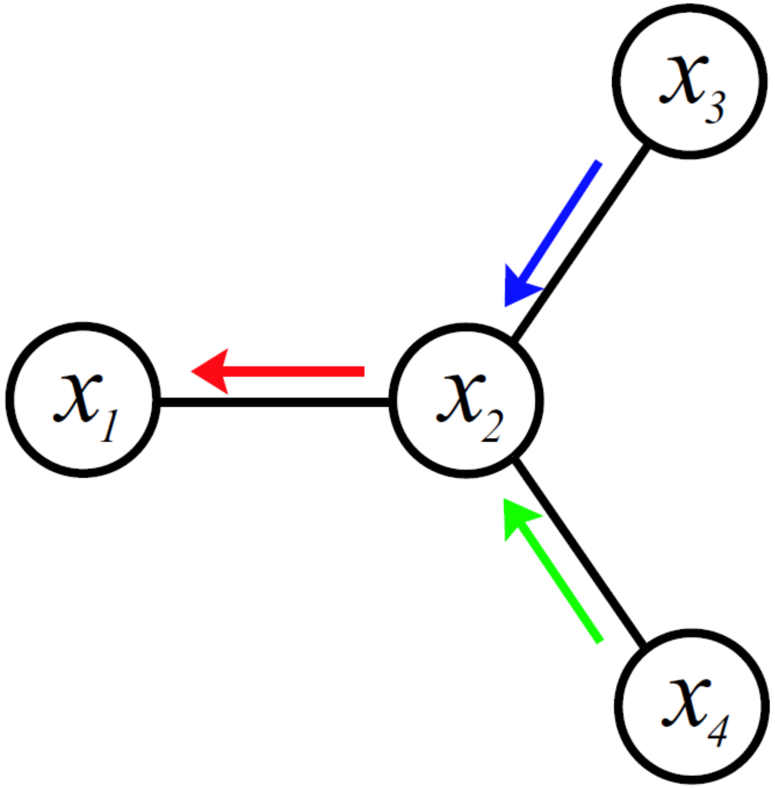
- Sum-Product Belief Propagation
- Loopy Belief Propagation
- Variable Elimination
- Junction Tree Algorithm
- Max-Product Belief Propagation

Outline

- ➢ Sum-Product Belief Propagation
- ➢ Loopy Belief Propagation
- ➢ Variable Elimination
- ➢ Junction Tree Algorithm
- ➢ Max-Product Belief Propagation

Why Graphical Models?

Structure simplifies both **representation** and **computation**



Representation

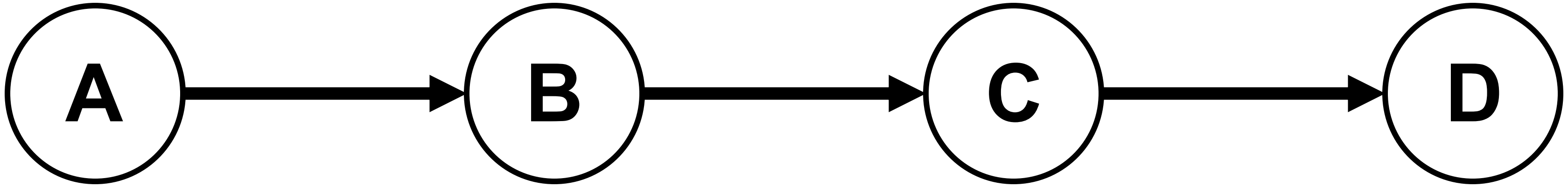
Complex global phenomena arise by simpler-to-specify local interactions

Computation

Inference / estimation depends only on subgraphs (e.g. dynamic programming, belief propagation, Gibbs sampling)

Example: Markov Chain

Suppose we have a chain graph…

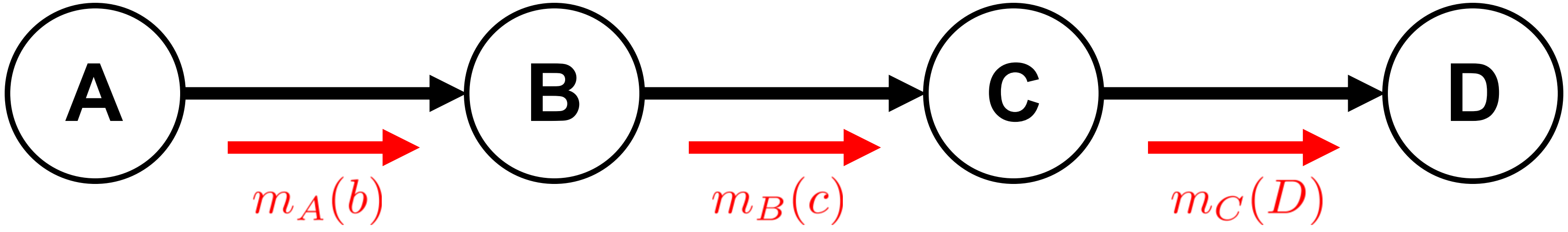


…and want to calculate the marginal on D

$$P(D) = \sum_a \sum_b \sum_c P(a, b, c, D)$$

- ➢ For K-valued variables this is $\mathcal{O}(K^3)$
- ➢ For a Markov Chain on N variables calculating $P(X_N)$ takes $\mathcal{O}(K^{N-1})$
- ➢ We can do better by reordering operations…

Example: Markov Chain



Suppose we just care about marginal on D:

$$P(D) = \sum_a \sum_b \sum_c P(a)P(b \mid a)P(c \mid b)P(D \mid c)$$

$$= \sum_c P(D \mid c) \sum_b P(c \mid b) \underbrace{\sum_a P(a)P(b \mid a)}$$

(Distributive property)

$$= \sum_c P(D \mid c) \underbrace{\sum_b P(c \mid b) m_A(b)}$$

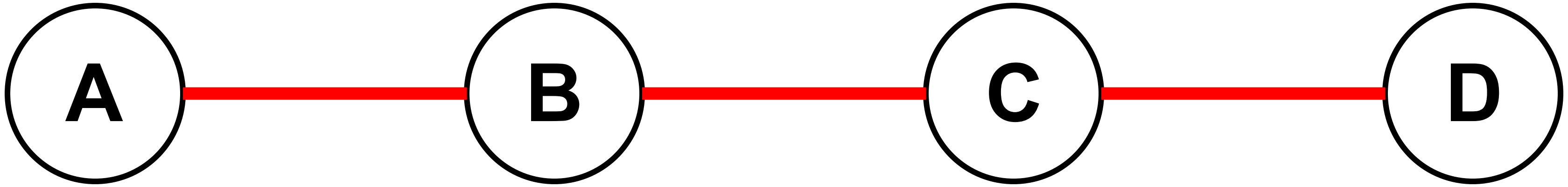
$$= \underbrace{\sum_c P(D \mid c) m_B(c)}$$

$$= m_C(D)$$

Each message takes O(K^2) time for total of O(3K^2)

On a Markov Chain of N RVs takes O((N-1)K^2)

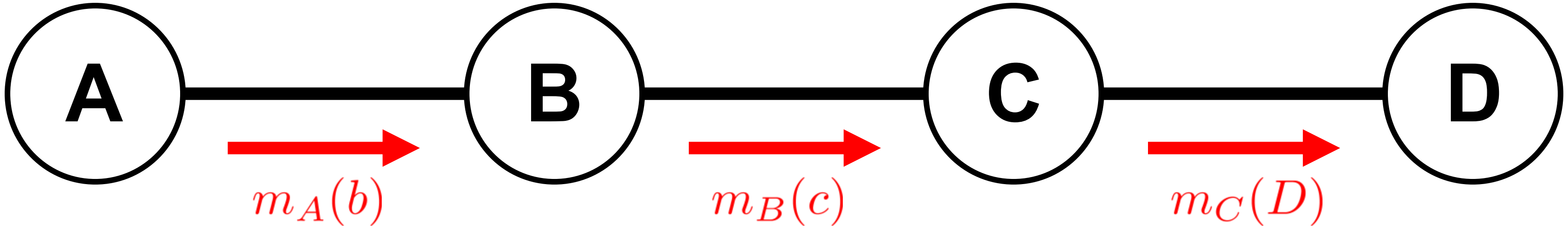
Example: Markov Chain



Convert Bayes net to MRF by ignoring local normalization:

$$P(A, B, C, D) \propto \psi(A)\psi(B, A)\psi(C, B)\psi(D, C)$$

Example: Markov Chain



Convert Bayes net to MRF by ignoring local normalization:

$$P(A,B,C,D) \propto \psi(A)\psi(B,A)\psi(C,B)\psi(D,C)$$

Repeat same procedure on MRF (we do not assume normalization):

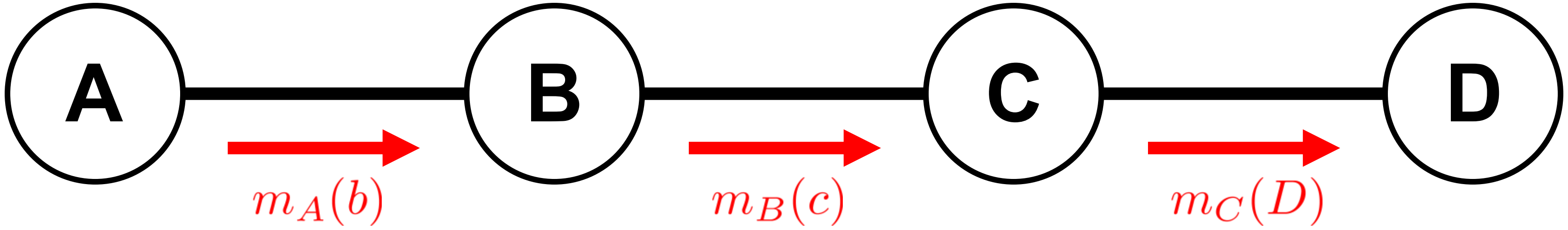
$$P(D) \propto \sum_c \psi(c,D) \sum_b \psi(b,c) \underbrace{\sum_a \psi(a,b)\psi(a)}$$

$$P(D) \propto \sum_c \psi(c,D) \sum_b \psi(b,c) m_A(b)$$

$$P(D) \propto \sum_c \psi(c,D) m_B(c)$$

$$P(D) \propto m_C(D)$$

Markov Chain Revisited



Inference viewed as passing *messages* e.g. C → D:

$$m_C(d) = \sum_c m_B(c)\psi(c,d)$$

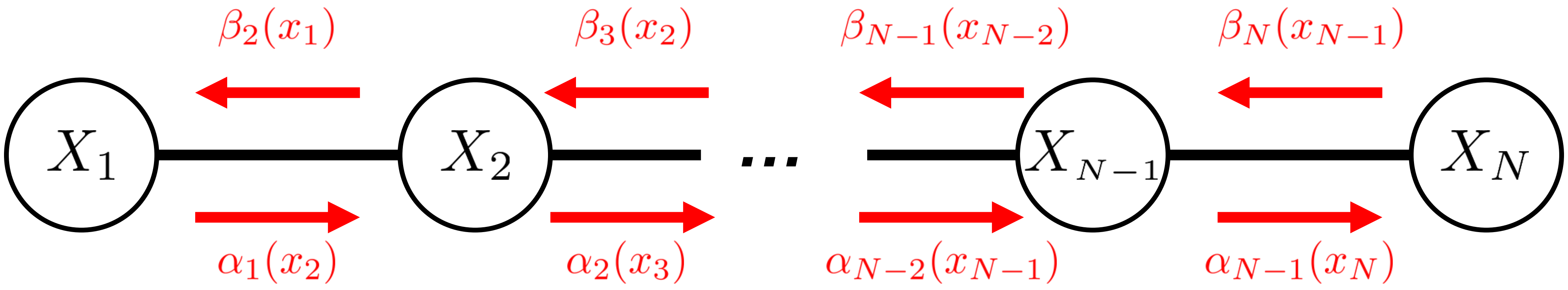
***Incoming* Message**

***Compatibility* Potential**

- ➢ Only showed calculation of marginal at rightmost node
- ➢ Backward pass of messages calculates *all* marginals
- ➢ General inference on Markov chains called *forward-backward* alg.
- ➢ Extension to other model structures called *sum-product algorithm*

Forward-Backward Algorithm

Pass messages forward/backward along chain…



Forward message:

$$\alpha_{n-1}(x_n) = \sum_{x_{n-1}} \alpha_{n-2}(x_{n-1})\psi(x_{n-1}, x_n)$$

Forward message:

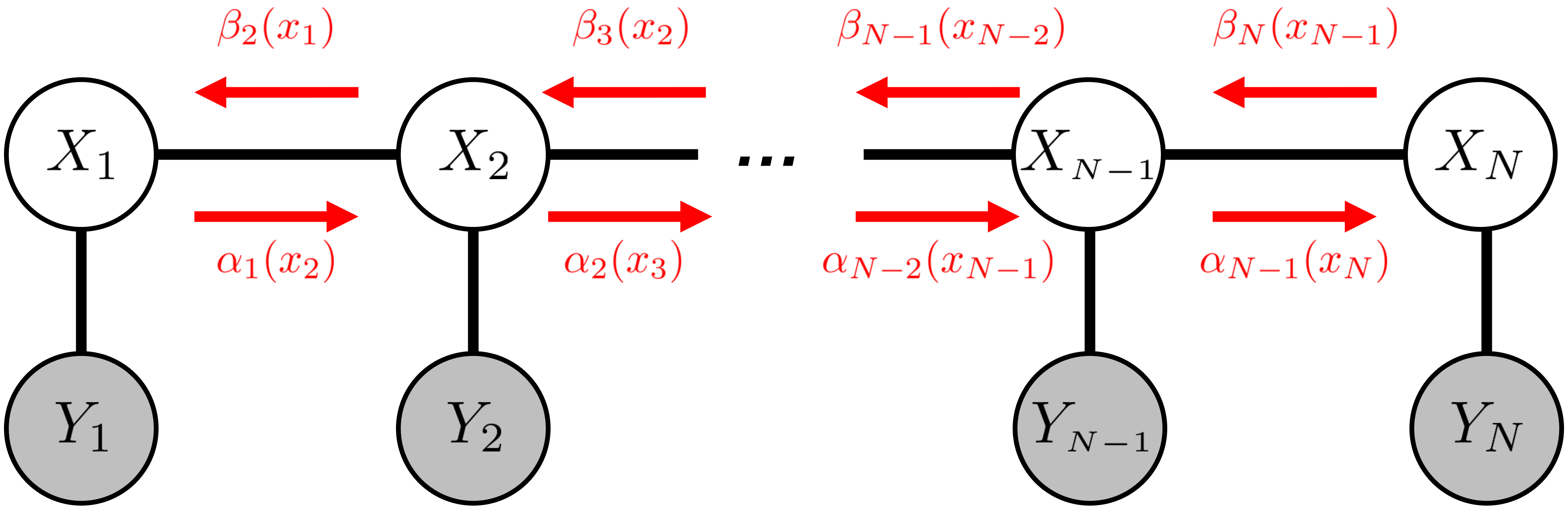
$$\beta_{n+1}(x_n) = \sum_{x_{n+1}} \beta_{n+2}(x_{n+1})\psi(x_n, x_{n+1})$$

Marginal probability:

$$p(x_n) \propto \alpha_{n-1}(x_n)\beta_{n+1}(x_n)$$

Forward-Backward Algorithm

Extends to HMM-style graphs with node observations…



Forward message:

$$\alpha_{n-1}(x_n) = \psi(x_n, y_n) \sum_{x_{n-1}} \alpha_{n-2}(x_{n-1}) \psi(x_{n-1}, x_n)$$

Backward message:

$$\beta_{n+1}(x_n) = \sum_{x_{n+1}} \beta_{n+2}(x_{n+1}) \psi(x_n, x_{n+1}) \psi(x_{n+1}, y_{n+1})$$

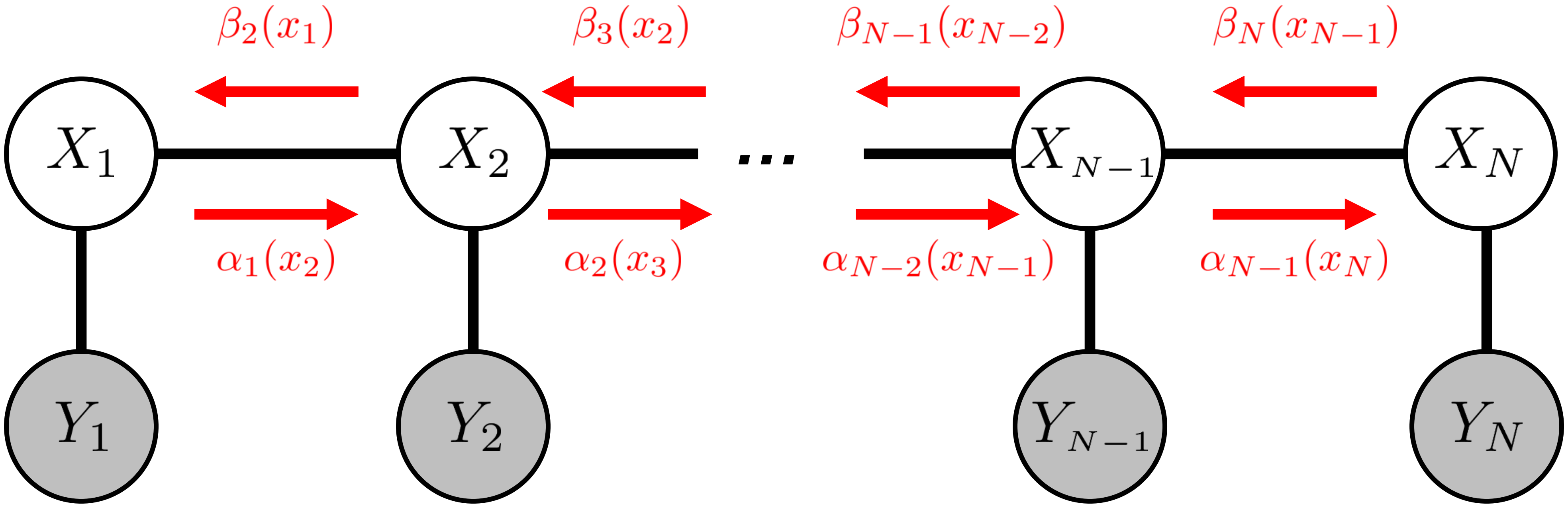
Forward-Backward Algorithm

$$
\begin{aligned}
\alpha_{n-1}(x_n) &\propto p(y_1, \ldots, y_n, x_n) \\
&= p(y_1, \ldots, y_n \mid x_n) p(x_n) && \textbf{(Chain rule)} \\
&= p(y_n \mid x_n) p(y_1, \ldots, y_{n-1} \mid x_n) p(x_n) && \textbf{(Conditional Independence)} \\
&= p(y_n \mid x_n) p(y_1, \ldots, y_{n-1}, x_n) && \textbf{(Chain rule)} \\
&= p(y_n \mid x_n) \sum_{x_{n-1}} p(y_1, \ldots, y_{n-1}, x_{n-1}, x_n) && \textbf{(Law of Total Probability)} \\
&= p(y_n \mid x_n) \sum_{x_{n-1}} p(y_1, \ldots, y_{n-1}, x_{n-1}) p(x_n \mid x_{n-1}) && \textbf{(Chain rule + Conditional Independence)} \\
&\propto \psi(y_n, x_n) \sum_{x_{n-1}} \alpha_{n-2}(x_{n-1}) \psi(x_n, x_{n-1})
\end{aligned}
$$

Forward-Backward Algorithm

$$
\begin{aligned}
\beta_{n+1}(x_n) &\propto p(y_{n+1}, \ldots, y_N \mid x_n) \\
&= \sum_{x_{n+1}} p(y_{n+1}, \ldots, y_N, x_{n+1} \mid x_n) && \textbf{(Law of Total Probability)} \\
&= \sum_{x_{n+1}} p(y_{n+1}, \ldots, y_N \mid x_n, x_{n+1}) p(x_{n+1} \mid x_n) && \textbf{(Chain rule)} \\
&= \sum_{x_{n+1}} p(y_{n+1}, \ldots, y_N \mid x_{n+1}) p(x_{n+1} \mid x_n) && \textbf{(Conditional Independence)} \\
&= \sum_{x_{n+1}} p(y_{n+2}, \ldots, y_N \mid x_{n+1}) p(y_{n+1} \mid x_{n+1}) p(x_{n+1} \mid x_n) && \textbf{(Chain rule)} \\
&\propto \sum_{x_{n+1}} \beta_{n+2}(x_{n+1}) \psi(x_{n+1}, y_{n+1}) \psi(x_n, x_{n+1})
\end{aligned}
$$

Forward-Backward Algorithm



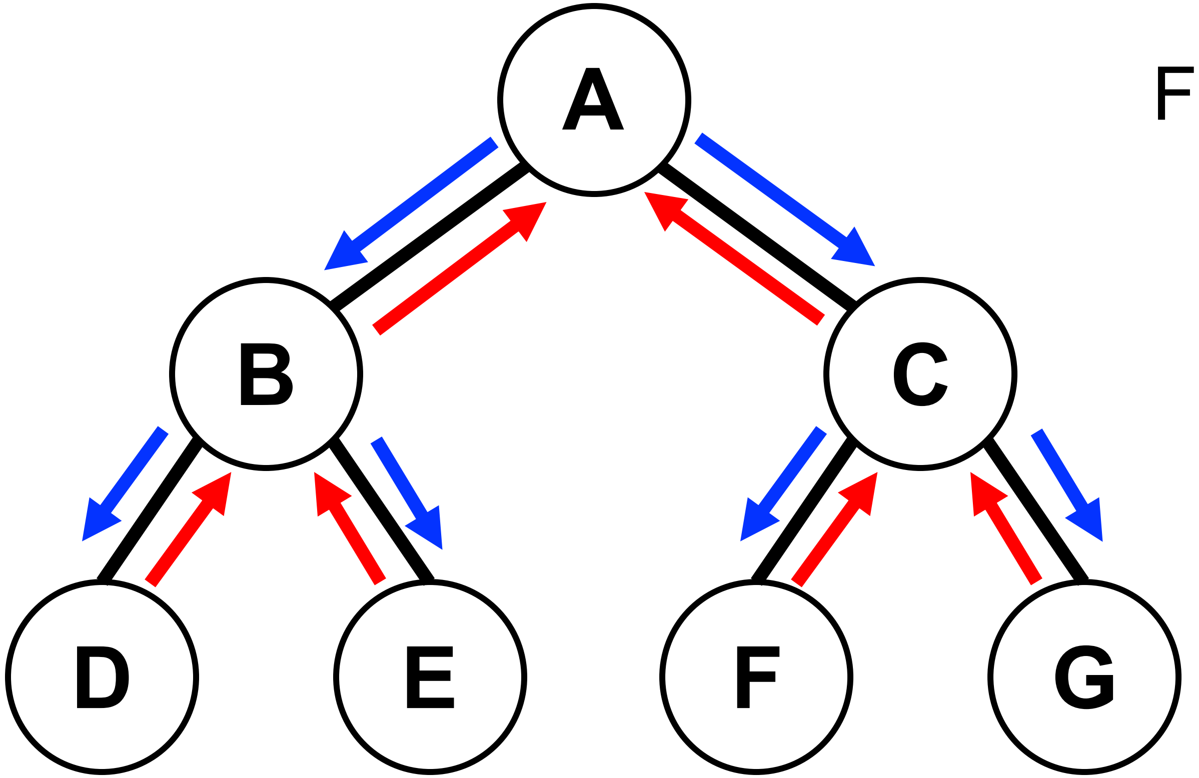
Forward message gives the *filtered posterior*:

$$\alpha_{n-1}(x_n) \propto p(y_1, \ldots, y_n, x_n) \propto p(x_n \mid y_1, \ldots, y_n)$$

Smoothed posterior incorporates all observations:

$$p(x_n \mid y_1, \ldots, y_N) \propto p(x_n \mid y_1, \ldots, y_n) p(y_{n+1}, \ldots, y_N \mid x_n)$$
$$\propto \alpha_{n-1}(x_n) \beta_{n+1}(x_n)$$

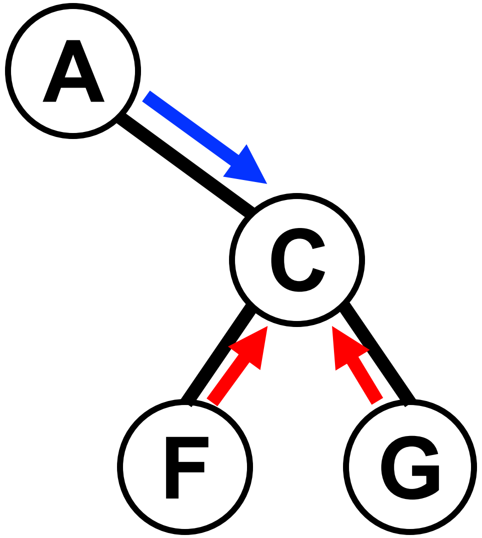
Sum-Product Belief Propagation



Pass messages from leaves-to-root, then root-to-leaves

Forward-Backward extends to *any tree-structured pairwise MRF*

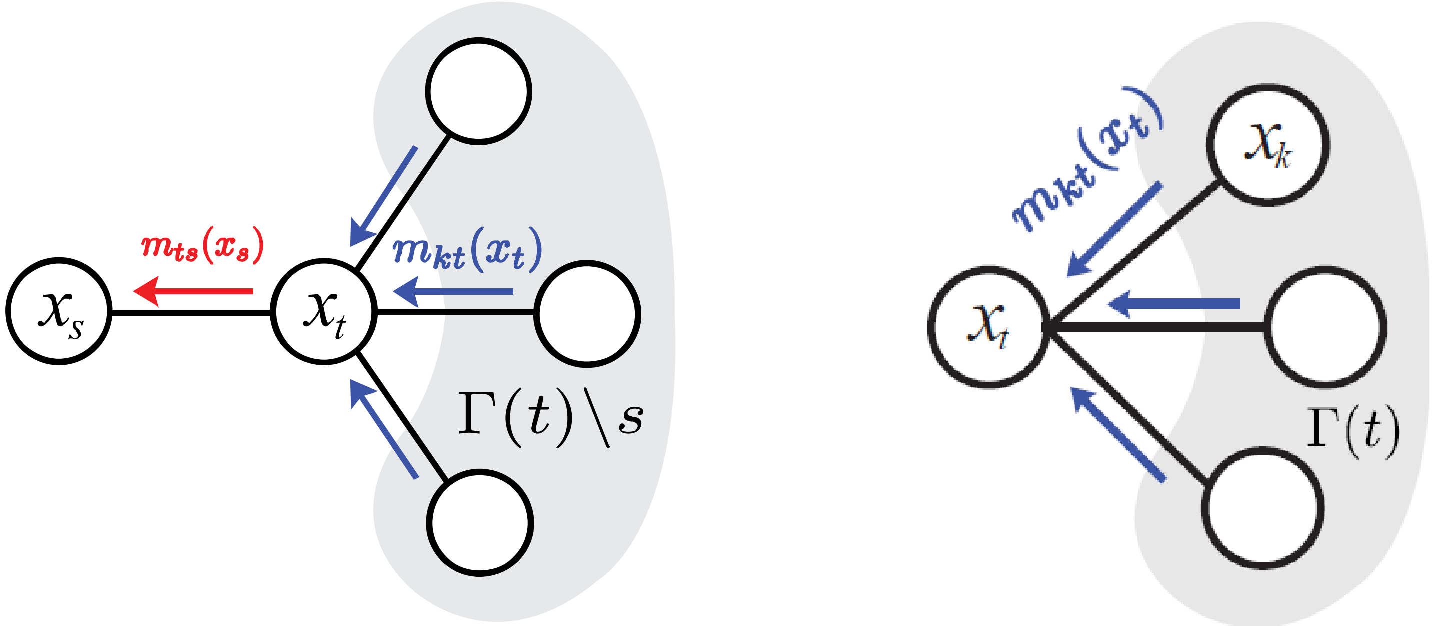
Marginal given by *incoming* messages (e.g. node C):



$$p(C) \propto \psi(C) m_A(C) m_F(C) m_G(C)$$

Sum-Product Belief Propagation

Message updates depend only on Markov blanket…



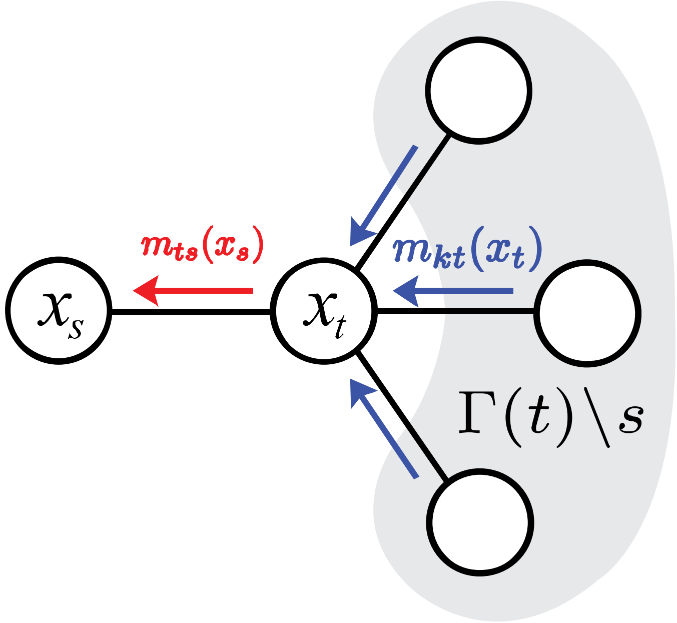
Message $m_{ts}(x_s) = \sum_{x_t} \psi_{st}(x_s, x_t)\psi_t(x_t) \prod_{k\in\Gamma(t)\backslash s} m_{kt}(x_t)$

Marginal $p(x_t) \propto \psi_t(x_t) \prod_{k\in\Gamma(t)} m_{kt}(x_t)$

Messages involve a **sum** over **products**, hence the name “sum-product algorithm”

Computational Complexity

$$m_{ts}(x_s) = \sum_{x_t} \underbrace{\psi_{st}(x_s, x_t)\psi_t(x_t) \prod_{k \in \Gamma(t) \setminus s} m_{kt}(x_t)}_{\phi(x_s, x_t)}$$



For K-valued random variables X_s and X_t intermediate factor $\psi(x_s, x_t)$ is K-by-K matrix

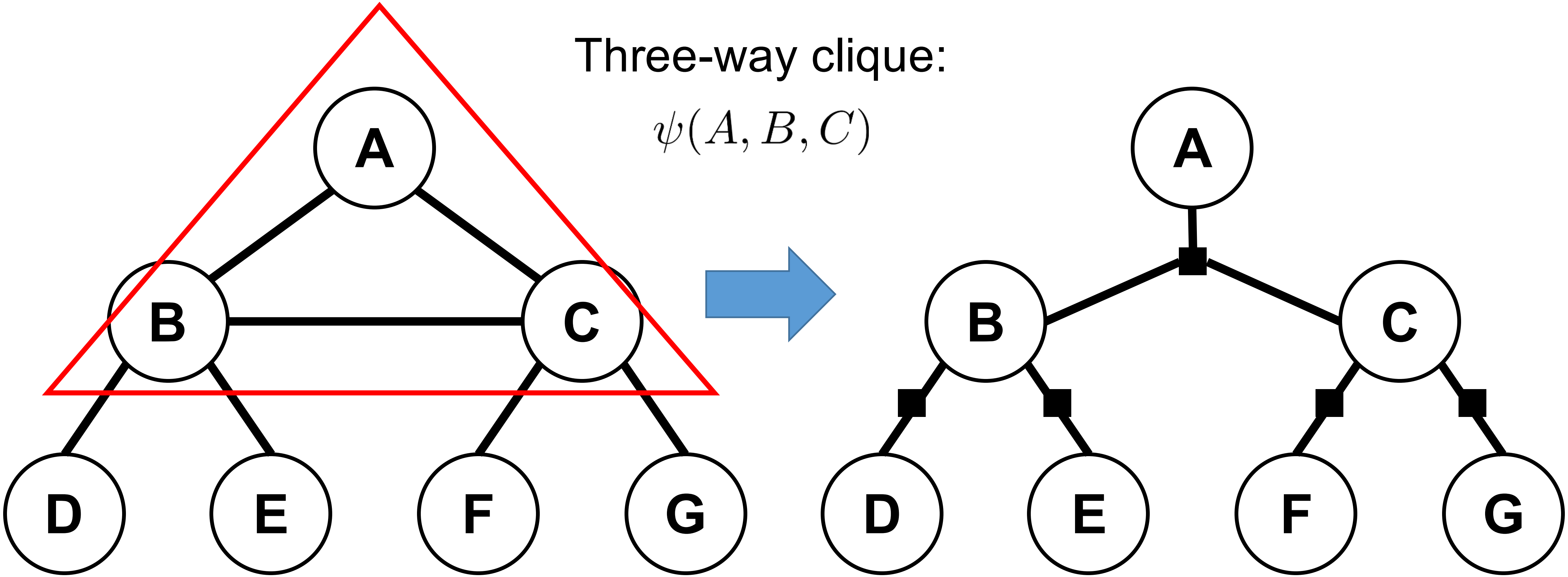
Each message requires computation:

$$\mathcal{O}(K^2)$$

There are |E| edges so total computation is:

$$\mathcal{O}(2|E|K^2)$$

Non-Pairwise MRFs



Three-way clique:

$\psi(A, B, C)$

Convert to tree-structured factor graph and redefine sum-product messages

Notation Change

We will use slightly different notation for this section…

Previous Notation

$\psi(x)$: Factors

$m(x)$: Messages

→

New Notation

$f(x)$: Factors

$\mu(x)$: Messages

Sum-Product Belief Propagation

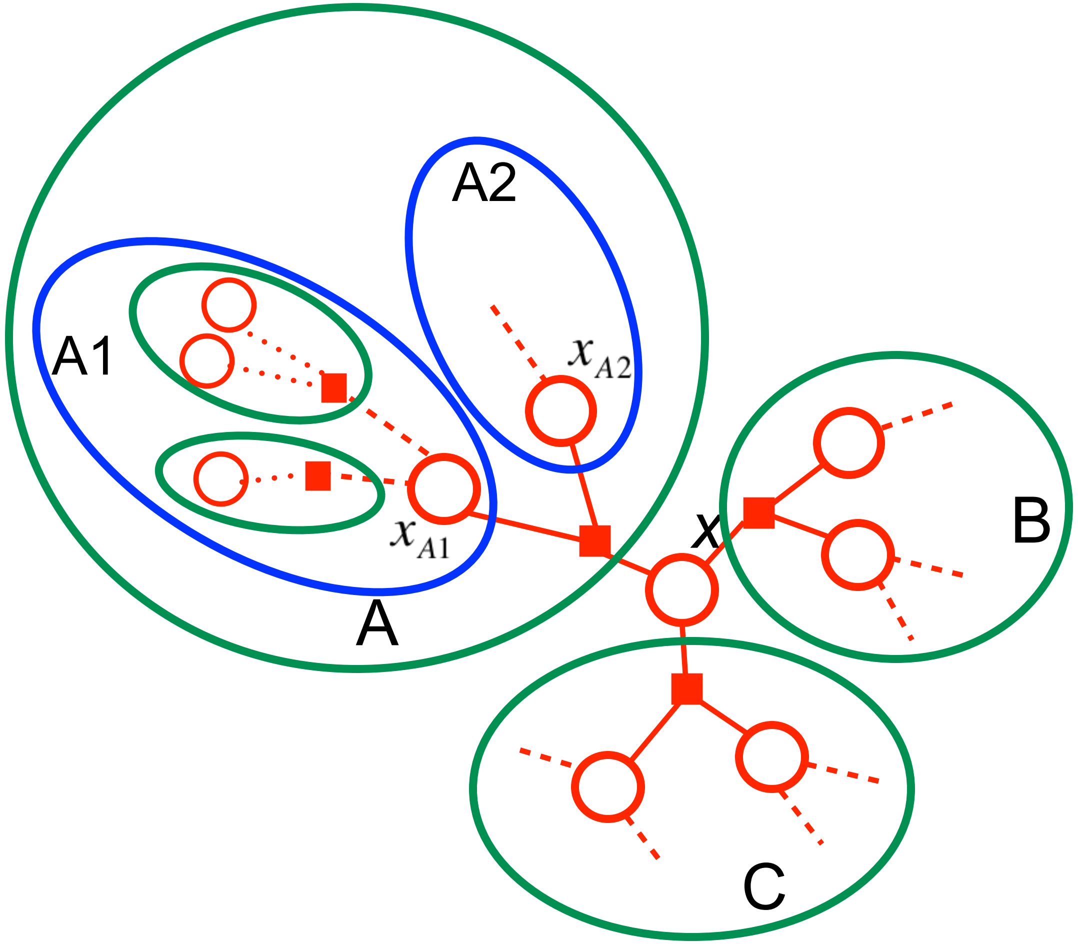
Sum-product extends to tree-structured *factor graphs*

Key Observation

Any variable node X with N factors splits graph into N subgraphs with no shared variables

Approach

Recursively decompose into subtrees and marginalize them

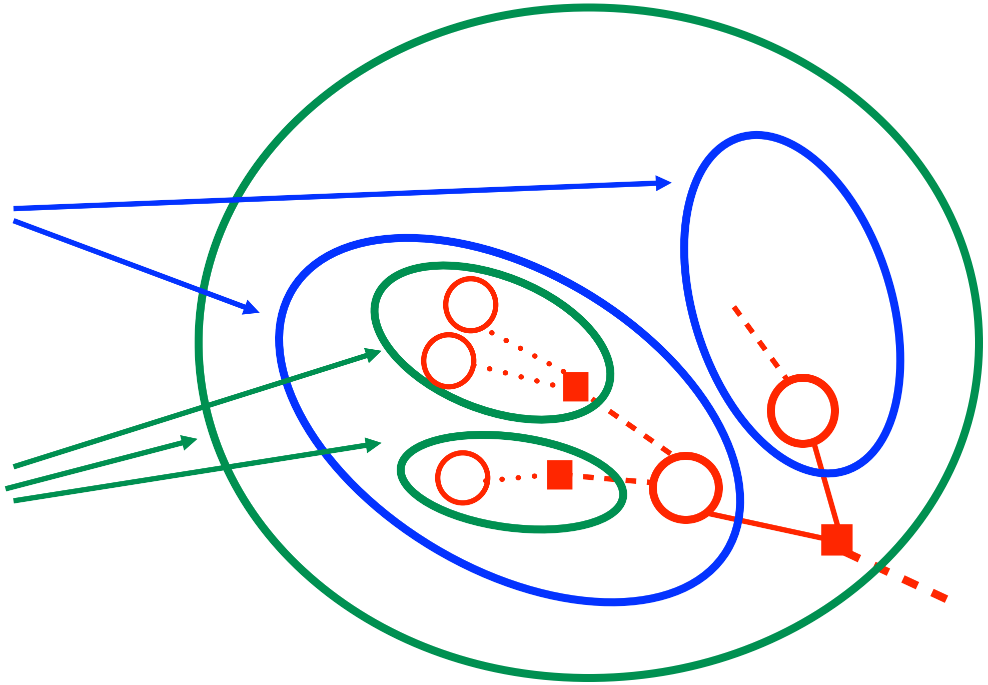


Sum-Product Belief Propagation

Two kinds of computations
marginalize different subtrees

Marginalize a sub-graph with a **variable node at its root** using the marginals of the sub-graphs attached to it.

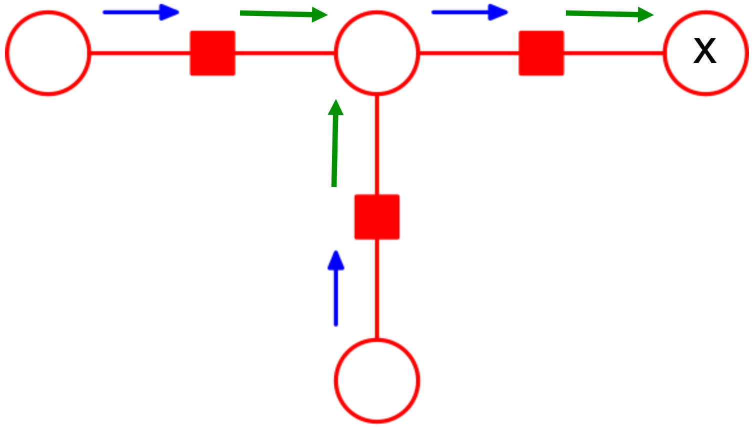
Marginalize a sub-graph with a **factor node at its root** using the marginals of the sub-graphs attached to it.



Each root node (variable or factor) “waits” for all messages from its children before being marginalized out

Sum-Product Belief Propagation

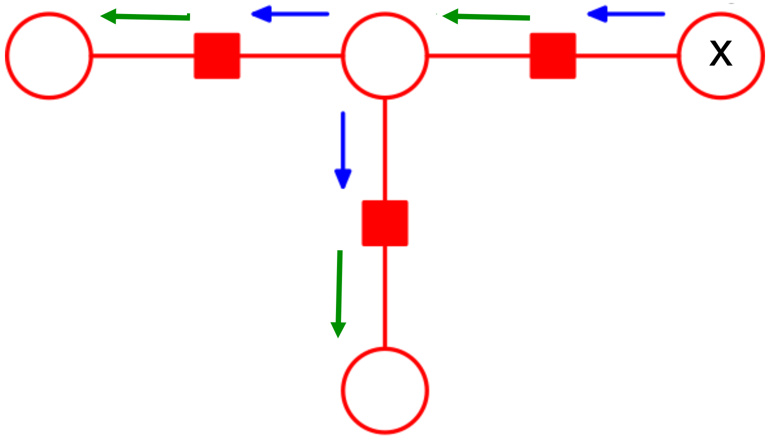
To the root (*x*)



Factor-to-variable

$\mu_{f \to x}$

From the root (*x*)



Variable-to-factor

$\mu_{x \to f}$

Factor-to-variable message

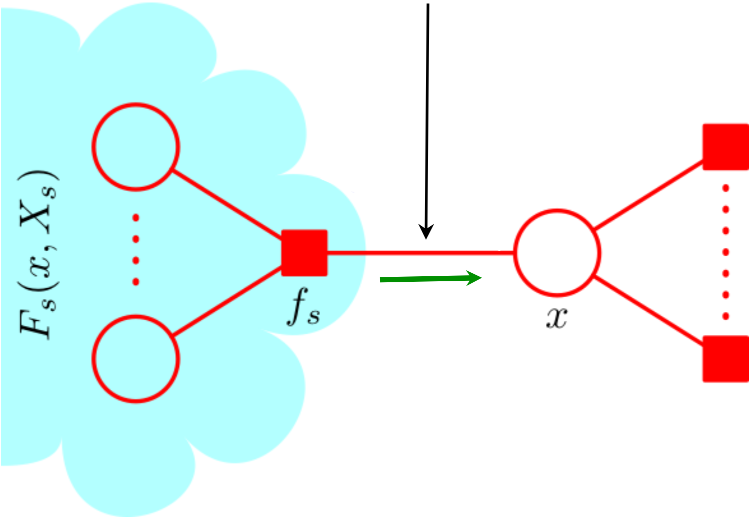
Let X_s be the variables of the sub-graph attached to a factor, f_s (as root).

Denote the distribution of the sub-graph by $F_s(x, X_s)$

Define the factor-to-variable message from f_s to x by:

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

The message is the marginal of the sub-graph with respect to all variables **except *x***.



Variable-to-factor message

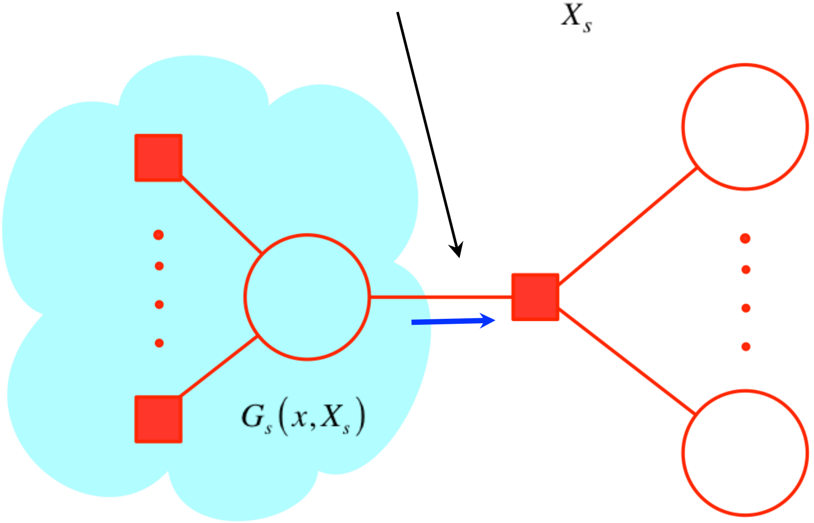
Let X_s be the variables in the sub-graph attached to a variable, x (as root).

Denote the distribution of the sub-graph by $\mathrm{G}_s(x, X_s)$

Define the variable-to-factor message from x to f_s by:

$$\mu_{x \to f_s}(x) \equiv \sum_{X_s} G_s(x, X_s)$$

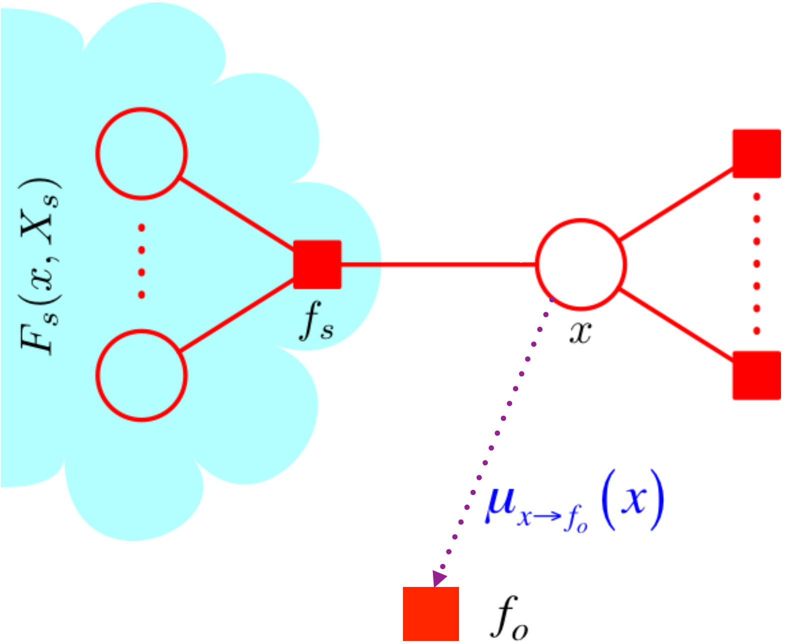
The message is the marginal of the sub-graph with respect to all variables **except *x***.



What a variable node computes

The outgoing message to the factor, f_o, from x, is exactly the same marginal as the previous, except we exclude f_o.

$$\mu_{x\to f_o}(x) = \sum_{\mathbf{x}/x} \prod_{s\in ne(x)/f_o} F(x, X_s)$$

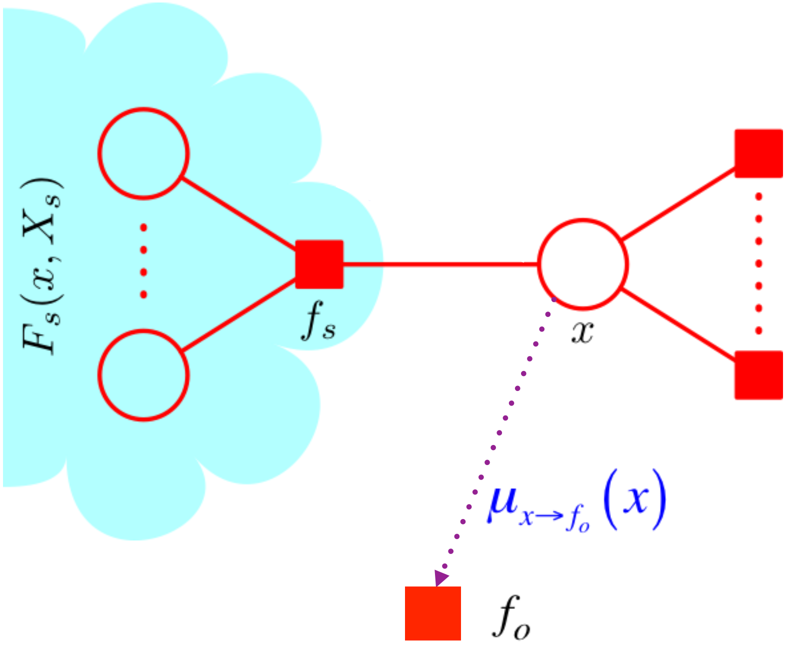


*This is **what** it computes, but not **how** it does it efficiently (i.e., as in the sum-product algorithm).

General variable node computation

The outgoing message to the factor, f_o, from x, is exactly the same marginal as the previous, except we exclude f_o.

$$\mu_{x\to f_o}(x) = \sum_{\mathbf{x}/x} \prod_{s\in ne(x)/f_o} F(x, X_s)$$



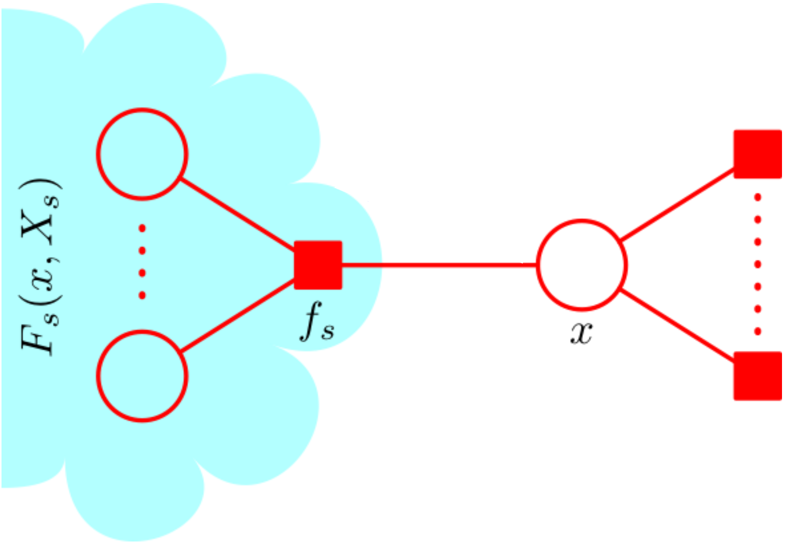
In the following, we will consider the first case, $\tilde{p}(x)$, but everything works the same for $\mu_{x\to f_o}(x)$.

What the **root** variable node computes

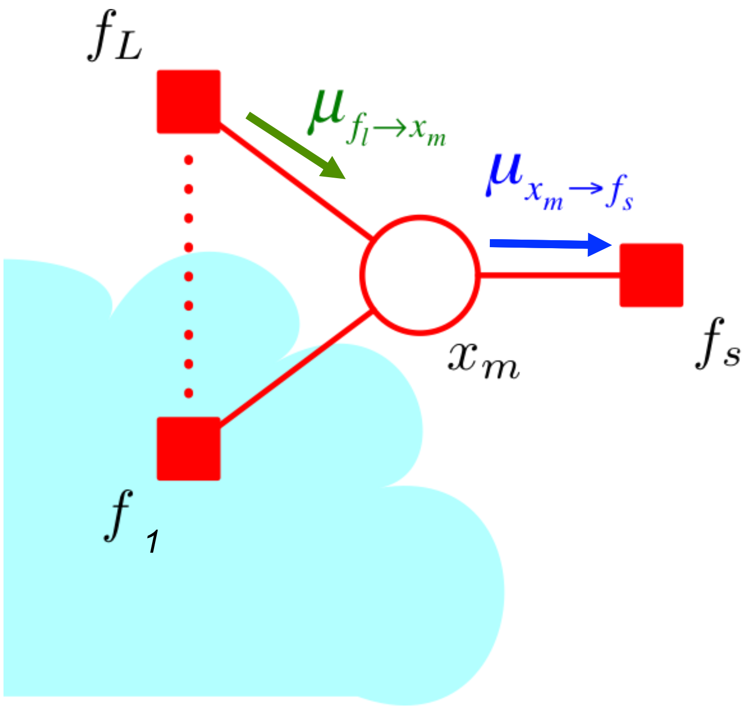
$$p(x) \propto \sum_{\mathbf{X}\setminus x} \underbrace{\prod_{s\in ne(x)} F_s(x, X_s)}$$

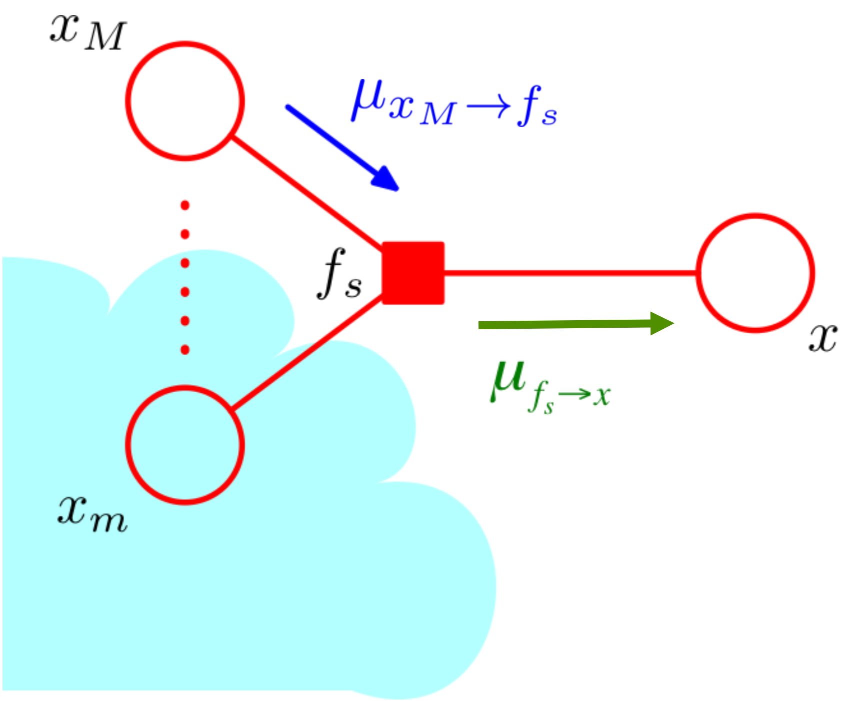
Product contains all factors in the graph with root *x*.

($ne(\bullet)$ denotes neighbours)



Sum-product on a slide





Variable node x_m gathers messages, $\mu_{f_l\to x_m}$, and sends

$$\mu_{x_m\to f_s}(x_m) = \prod_{l \ni f_l \in n(x_m)\setminus f_s} \mu_{f_l\to x_m}(x_m)$$

Factor f_s gathers messages $\mu_{x_m\to f_s}(x_m)$, and sends

$$\mu_{f_s\to x}(x) = \sum_{x_1}\sum_{x_2}\cdots\sum_{x_M} f_s(x,x_1,x_2,\ldots,x_M)\prod_{m\in ne(f_s)\setminus x}\mu_{x_m\to f_s}(x_m)$$

Marginal is product of incoming factor-to-variable messages:

$$p(x_m) \propto \prod_{f_l\in ne(x_m)} \mu_{f_l\to x_m}(x_m)$$

One point of confusion

The two products over messages look similar, but the **first**:

Variable node x_m gathers messages, $\mu_{f_l \to x_m}$, and sends

$$\mu_{x_m \to f_s}(x_m) = \boxed{\prod_{l \ni f_l \in n(x_m) \setminus f_s} \mu_{f_l \to x_m}(x_m)}$$

is a product of vectors, each over the same variable, but the **second** has the variable as the index in the product:

Factor f_s gathers messages $\mu_{x_m \to f_s}(x_m)$, and sends

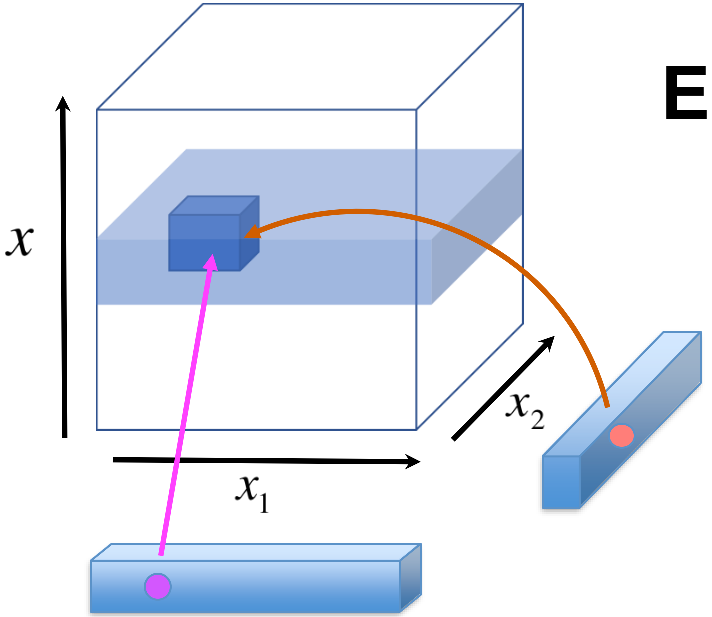
$$\mu_{f_s \to x}(x) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_M} f_s(x, x_1, x_2, \ldots, x_M) \boxed{\prod_{m \in ne(f_s) \setminus x} \mu_{x_m \to f_s}(x_m)}$$

One point of confusion (continued)

There are several ways to interpret the message product:

$$\mu_{f_s \to x}(x) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_M} f_s(x, x_1, x_2, \ldots, x_M) \boxed{\prod_{m \in ne(f_s) \setminus x} \mu_{x_m \to f_s}(x_m)}$$

N-dimensional analogue of the outer product creates a tensor:



E.g. For two messages each element of the sum corresponding to (x, x_1, x_2) is

$$f(x, x_1, x_2) \cdot \mu_1(x_1) \cdot \mu_2(x_2)$$

Computational Complexity

Factor f_s gathers messages $\mu_{x_m \to f_s}(x_m)$, and sends

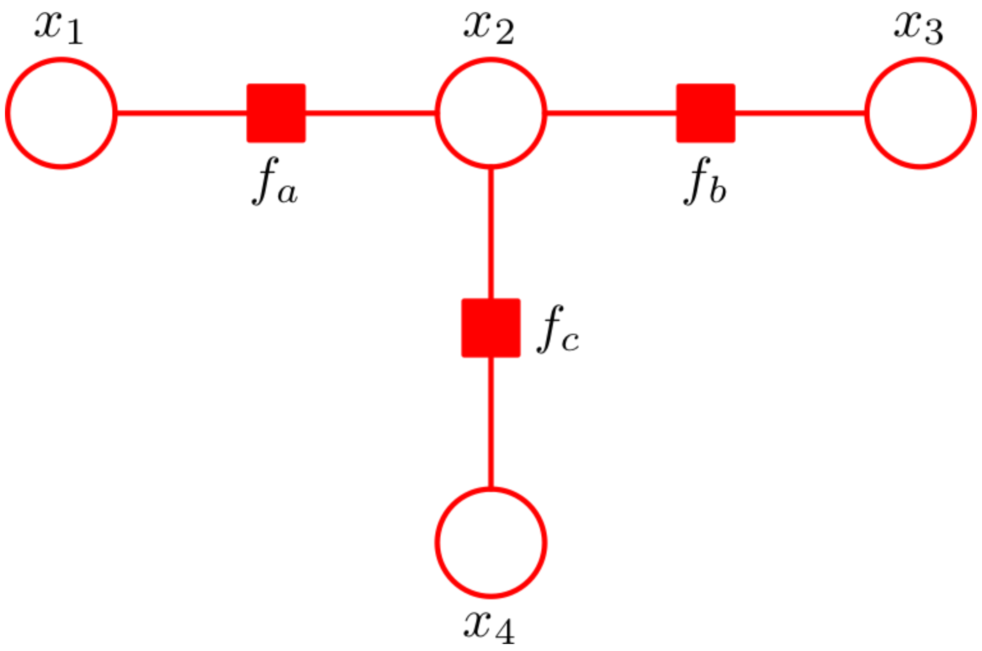
$$\mu_{f_s \to x}(x) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_M} \underbrace{f_s(x, x_1, x_2, \ldots, x_M) \prod_{m \in ne(f_s) \setminus x} \mu_{x_m \to f_s}(x_m)}_{\textbf{Intermediate factor}}$$

$$\phi(x, x_1, x_2, \ldots, x_M)$$

Assuming all variables are K-valued, intermediate factor with M+1 variables has $\mathcal{O}(K^{M+1})$ entries

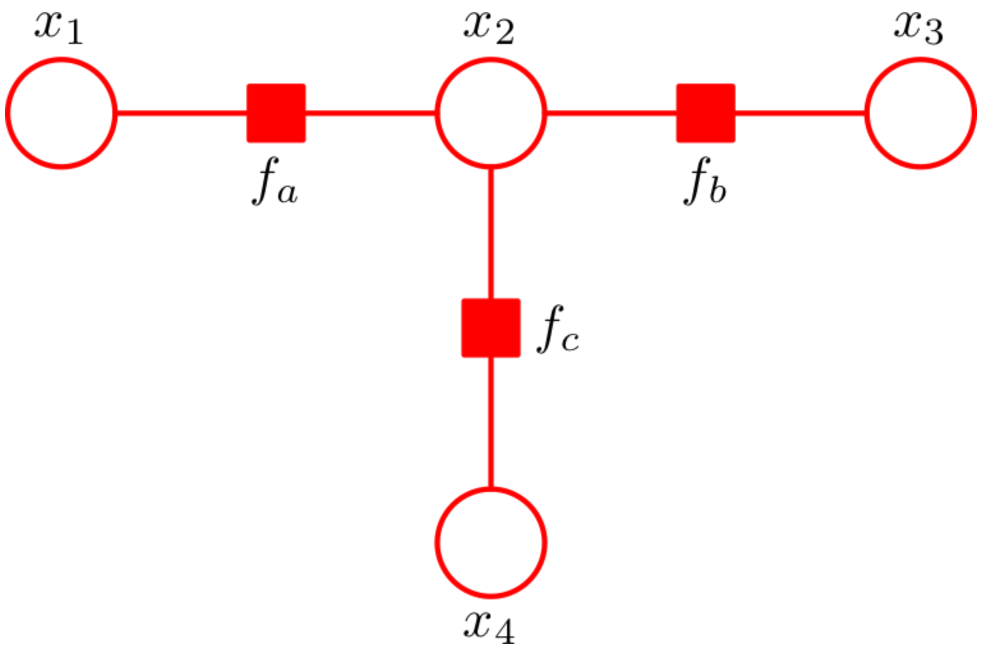
Sum-product algorithm example

Let $\tilde{\mathrm{p}}(\mathbf{x}) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4)$



Sum-product algorithm example

Let $\tilde{p}(\mathbf{x}) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4)$



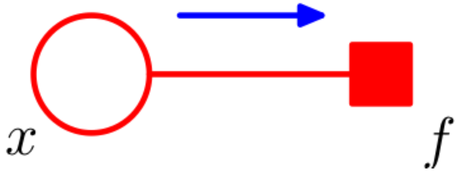
Declare x_3 as root node.

The sum-product algorithm

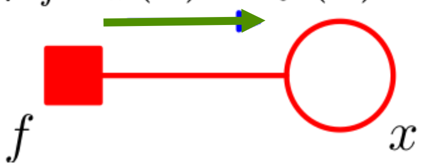
First, pass messages from leaves to your chosen root node. If you want more than one marginal or plan to do other computation, store the results as you go.

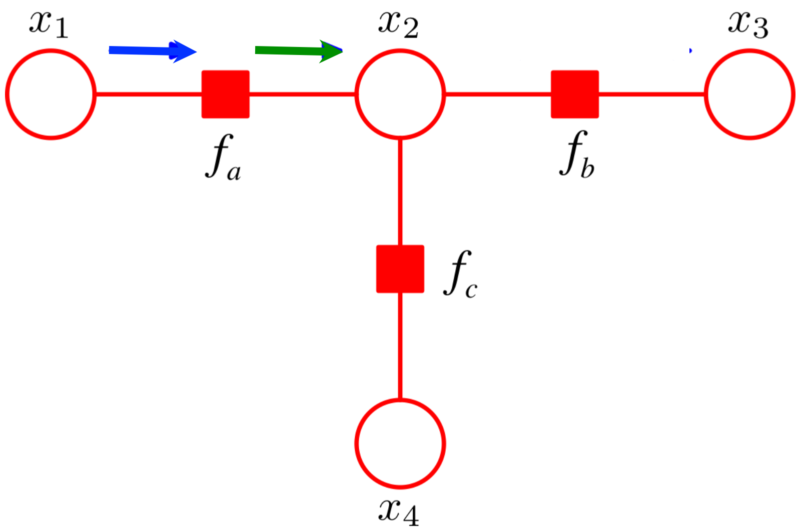
Initialization: If leaf node is a variable node, then start with a unity message. If leaf node is factor, then start with the factor.

$$\mu_{x \to f}(x) = 1$$



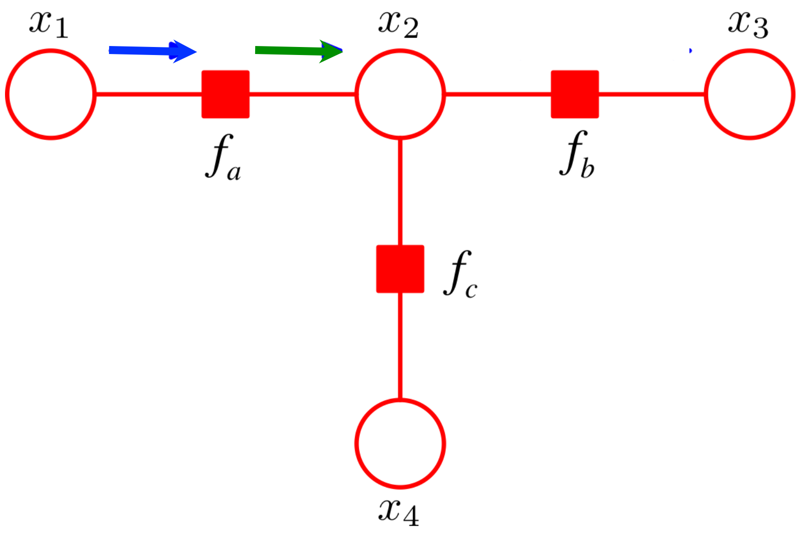
$$\mu_{f \to x}(x) = f(x)$$





$$\mu_{x_1 \to f_a}(x_1) = 1$$

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



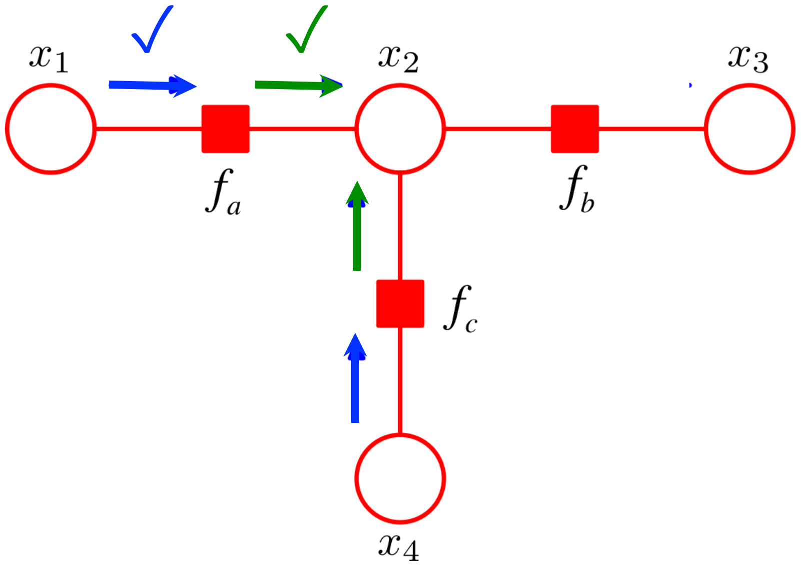
$$\mu_{x_1 \to f_a}(x_1) = 1$$

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

Recall the general case (don't confuse general variables with this example)

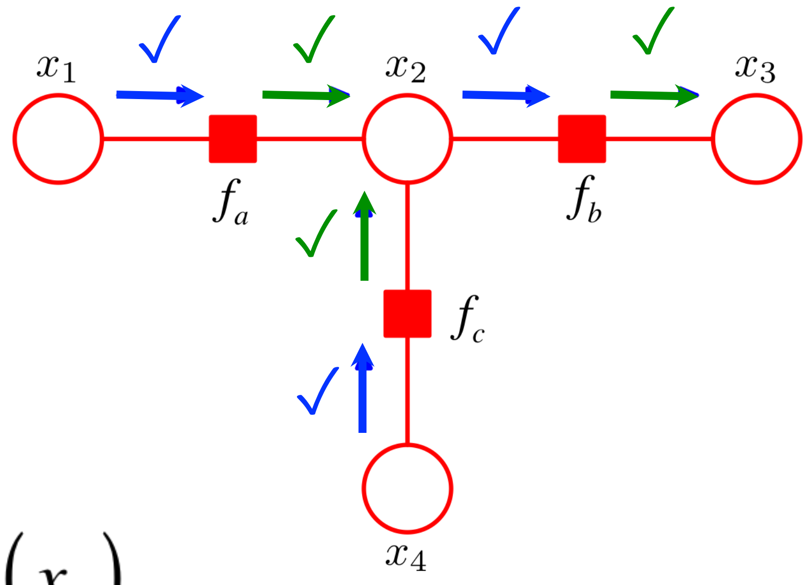
Factor f_s gathers messages $\mu_{x_m \to f_s}(x_m)$, and sends

$$\mu_{f_s \to x}(x) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_M} f_s(x, x_1, x_2, \ldots, x_M) \prod_{m \in ne(f_s) \setminus x} \mu_{x_m \to f_s}(x_m)$$



$$\mu_{x_4 \to f_c}(x_4) = 1$$

$$\mu_{f_c \to x_2}(x_2) = \sum_{x_4} f_c(x_2, x_4)$$



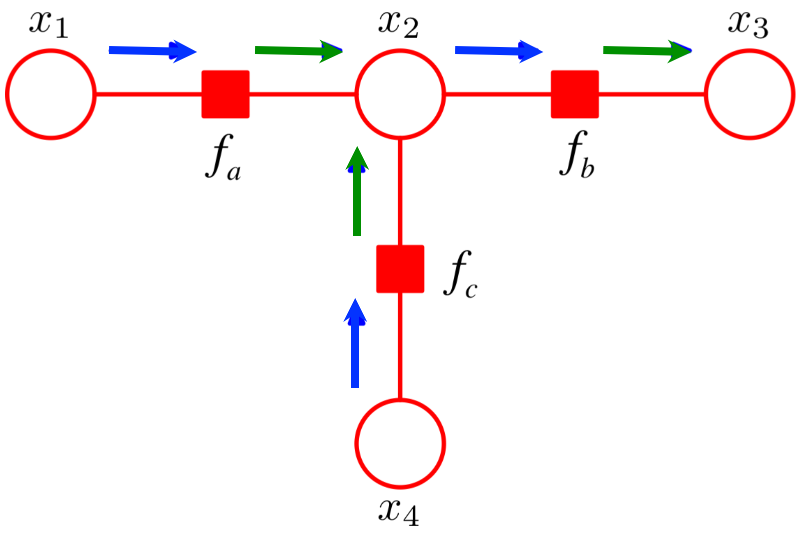
$$\mu_{x_2 \to f_b}(x_2) = \mu_{f_a \to x_2}(x_2)\,\mu_{f_c \to x_2}(x_2)$$

$$\mu_{f_b \to x_3}(x_3) = \sum_{x_2} f_b(x_2, x_3)\,\mu_{x_2 \to f_b}(x_2)$$

We now have the marginal at X_3:

$$p(x_3) \propto \mu_{f_b \to x_3}(x_3)$$

Summary of messages from leaves to root



$$\mu_{x_1 \to f_a}(x_1) = 1$$

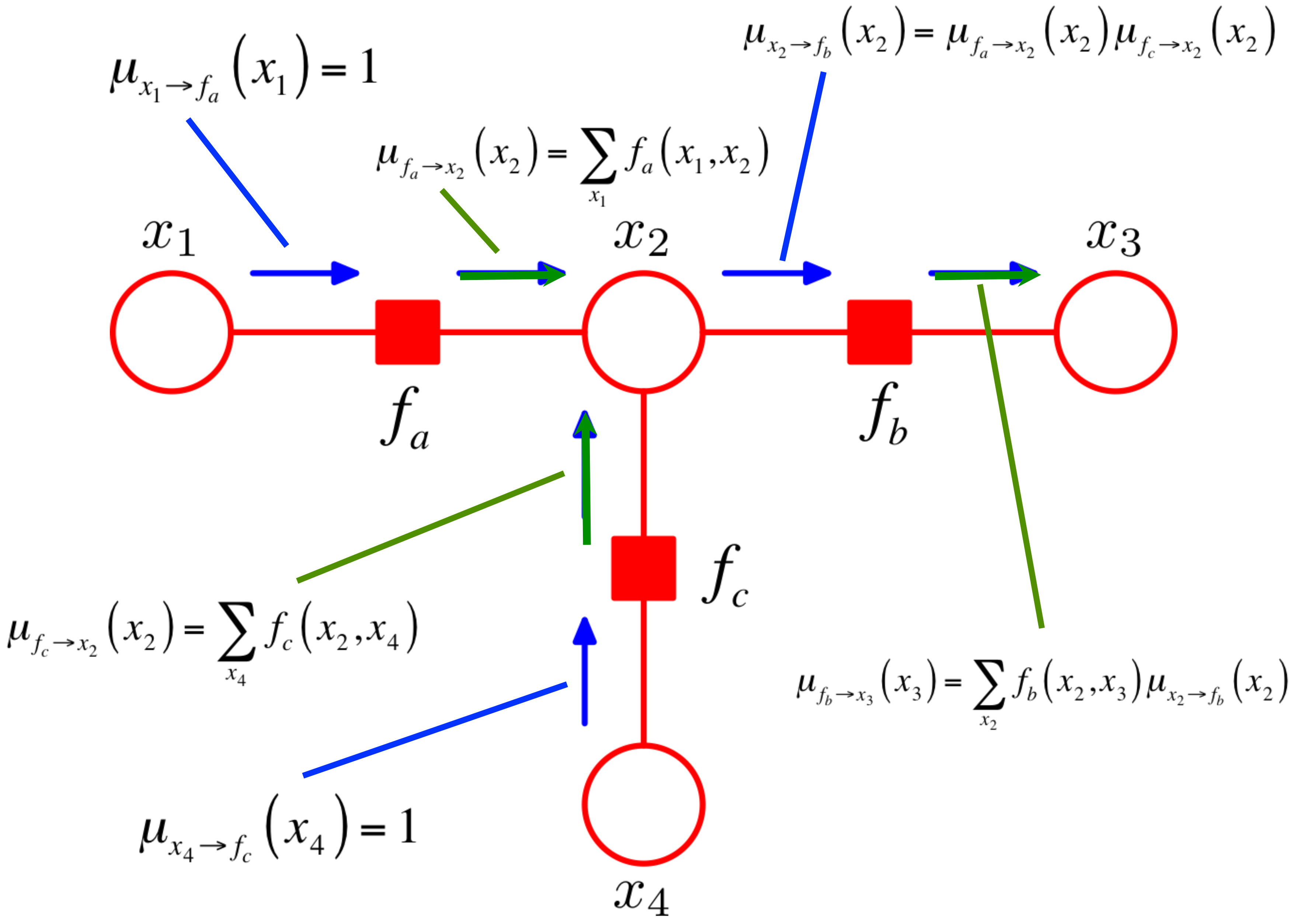
$$\mu_{f_a \to x_2}(x_2) = \sum_{x_1} f_a(x_1, x_2)$$

$$\mu_{x_4 \to f_c}(x_4) = 1$$

$$\mu_{f_c \to x_2}(x_2) = \sum_{x_4} f_c(x_2, x_4)$$

$$\mu_{x_2 \to f_b}(x_2) = \mu_{f_a \to x_2}(x_2)\mu_{f_c \to x_2}(x_2)$$

$$\mu_{f_b \to x_3}(x_3) = \sum_{x_2} f_b(x_2, x_3)\mu_{x_2 \to f_b}(x_2)$$



$$\mu_{x_1\to f_a}(x_1)=1$$

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

$$\mu_{x_2\to f_b}(x_2)=\mu_{f_a\to x_2}(x_2)\mu_{f_c\to x_2}(x_2)$$

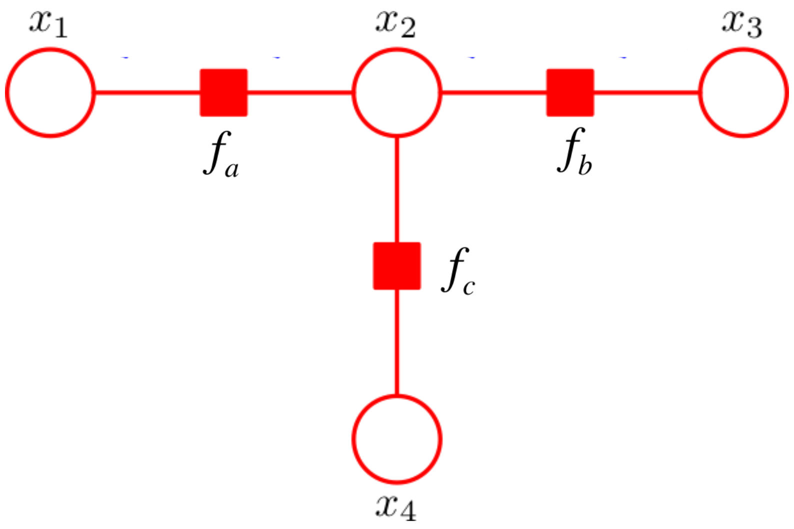
$$\mu_{f_c\to x_2}(x_2)=\sum_{x_4} f_c(x_2,x_4)$$

$$\mu_{f_b\to x_3}(x_3)=\sum_{x_2} f_b(x_2,x_3)\mu_{x_2\to f_b}(x_2)$$

$$\mu_{x_4\to f_c}(x_4)=1$$

Next we want to set up for additional computations, we pass messages from root to leaves.

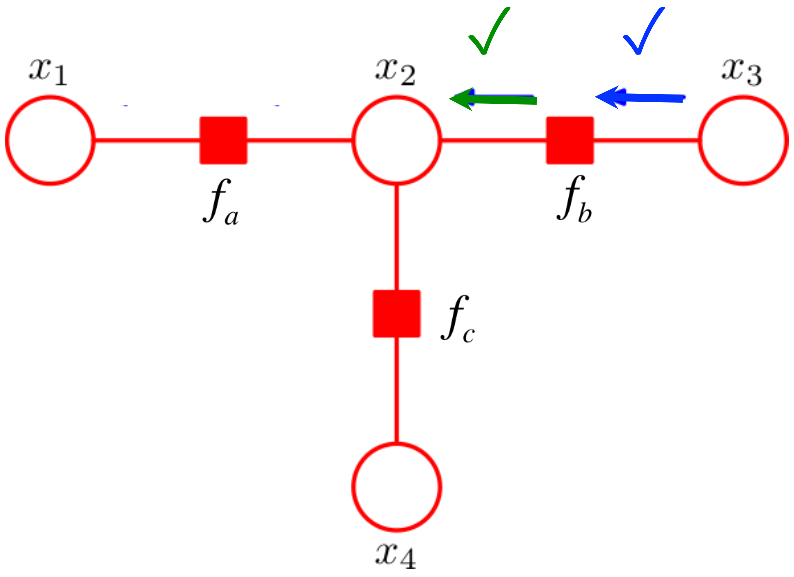
Candidate for the first and second ones?



Passing messages from root to leaves.

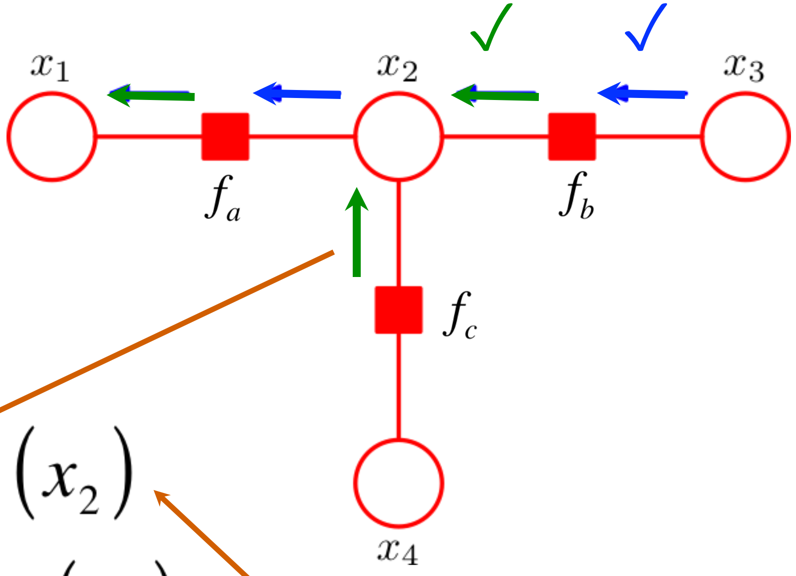
$$\mu_{x_3 \to f_b}(x_3) = 1$$

$$\mu_{f_b \to x_2}(x_2) = \sum_{x_3} f_b(x_2, x_3)$$



Candidate for third and fourth?

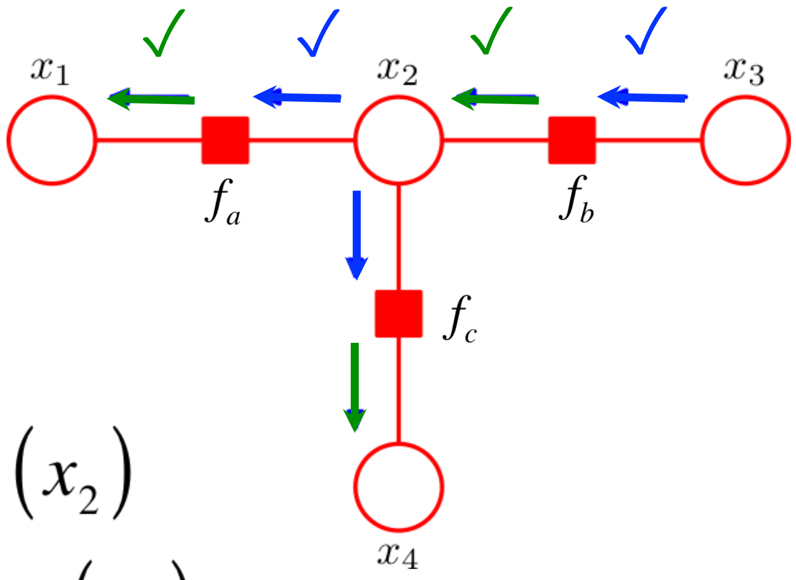
Lets go towards *x₁* first.



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

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

Note use of saved message from going the other way.

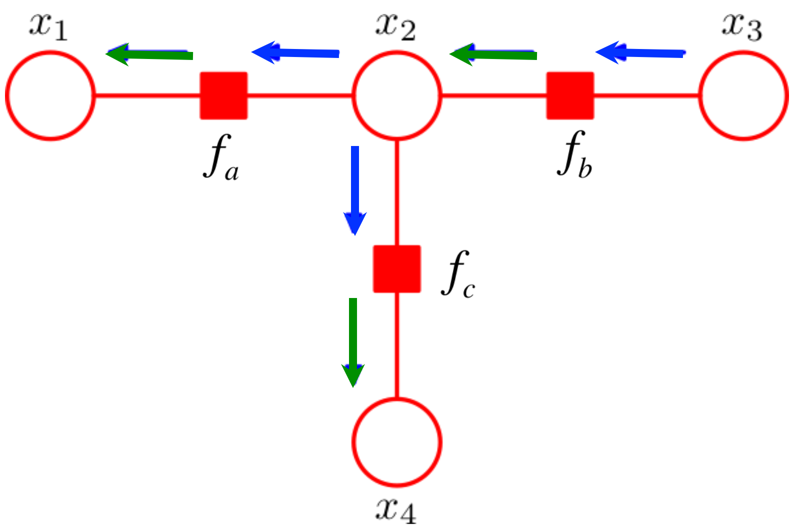


$$\mu_{x_2 \to f_c}(x_2) = \mu_{f_a \to x_2}(x_2)\,\mu_{f_b \to x_2}(x_2)$$

$$\mu_{f_c \to x_4}(x_4) = \sum_{x_2} f_c(x_2, x_4)\,\mu_{x_2 \to f_c}(x_2)$$

(similar to previous one)

Summary of messages from root to leaves.



$$\mu_{x_3 \to f_b}(x_3) = 1$$

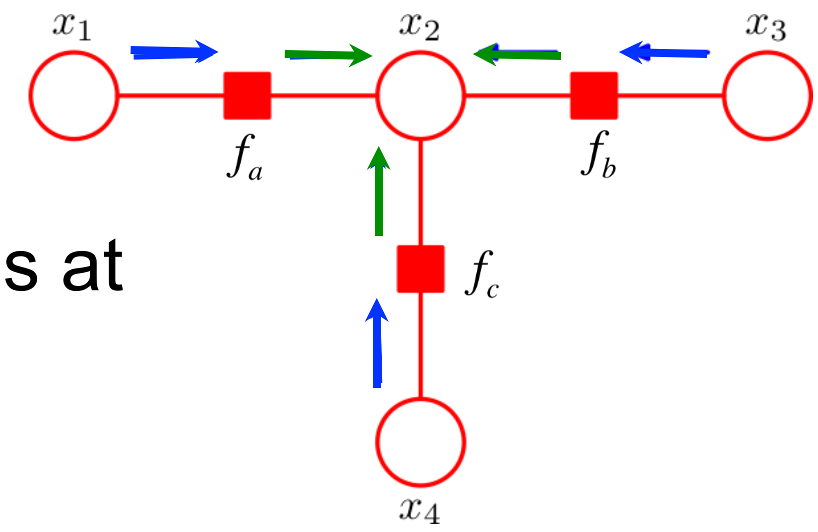
$$\mu_{f_b \to x_2}(x_2) = \sum_{x_3} f_b(x_2, x_3)$$

$$\mu_{x_2 \to f_a}(x_2) = \mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2)$$

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

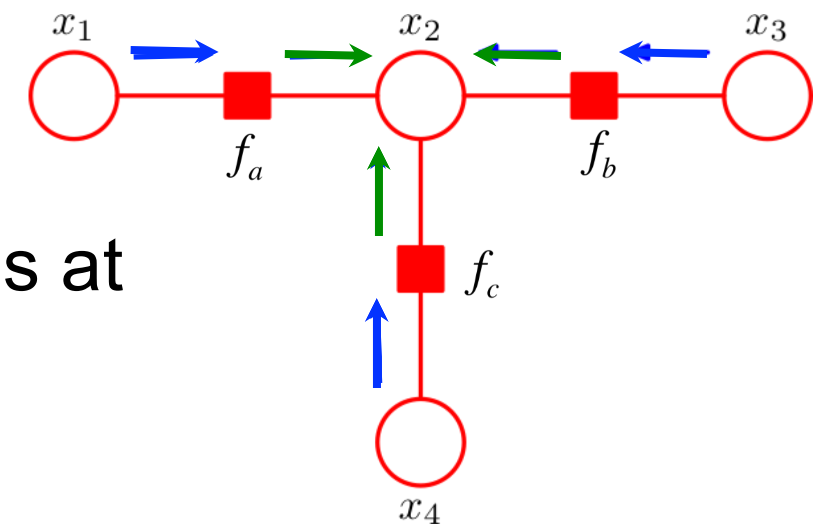
$$\mu_{x_2 \to f_c}(x_2) = \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)$$

$$\mu_{f_c \to x_4}(x_4) = \sum_{x_2} f_c(x_2, x_4)\mu_{x_2 \to f_c}(x_2)$$



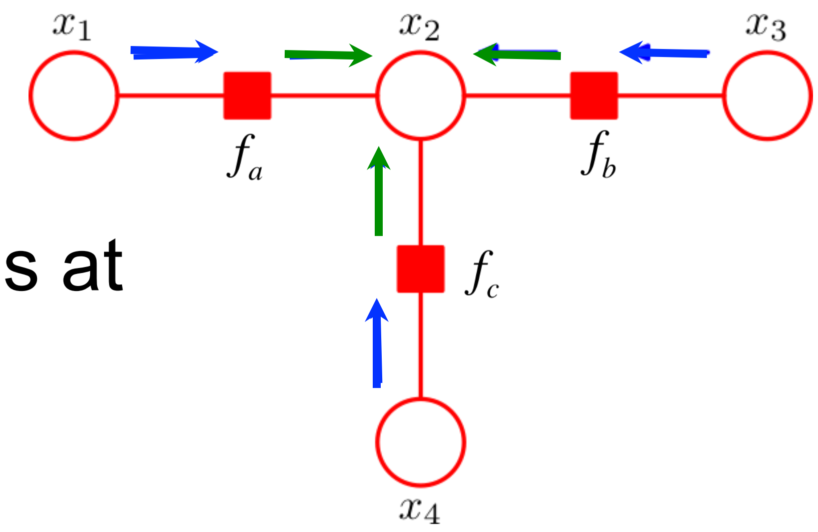
We can now compute marginals at *any* variable, e.g. X_2:

$$\tilde{p}(x_2) = \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2)$$



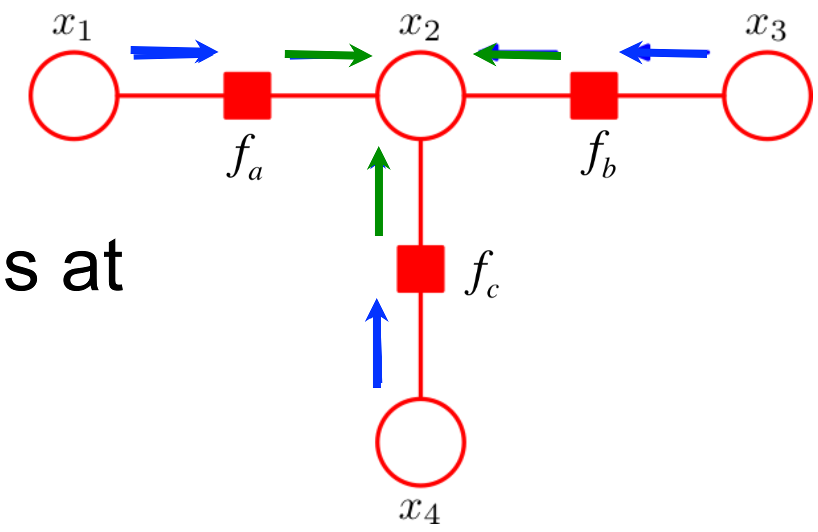
We can now compute marginals at *any* variable, e.g. X$_2$:

$$\begin{aligned}\tilde{p}(x_2) &= \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2) \\ &= \left(\sum_{x_1} f_a(x_1,x_2)\mu_{x_1 \to f_a}(x_1)\right)\left(\sum_{x_3} f_b(x_2,x_3)\mu_{x_3 \to f_b}(x_1)\right)\left(\sum_{x_4} f_c(x_2,x_4)\mu_{x_4 \to f_c}(x_1)\right)\end{aligned}$$



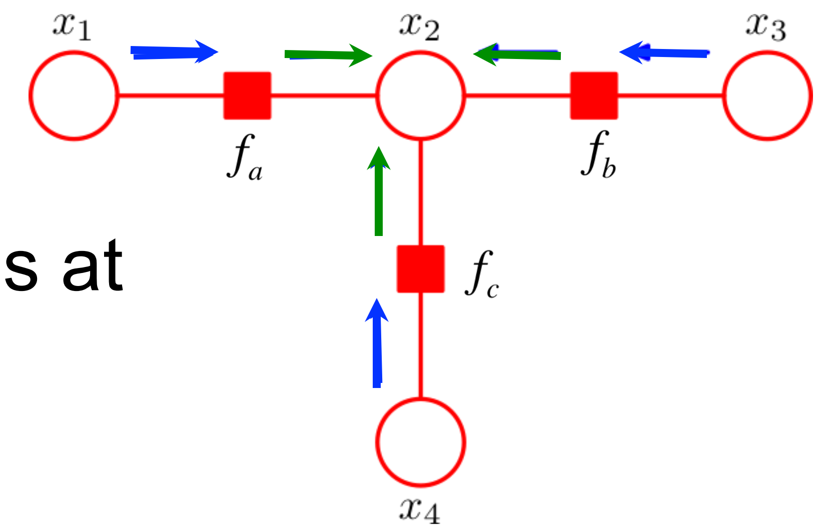
We can now compute marginals at *any* variable, e.g. X_2:

$$\begin{aligned}\tilde{p}(x_2) &= \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2) \\ &= \left(\sum_{x_1} f_a(x_1,x_2)\mu_{x_1 \to f_a}(x_1)\right)\left(\sum_{x_3} f_b(x_2,x_3)\mu_{x_3 \to f_b}(x_1)\right)\left(\sum_{x_4} f_c(x_2,x_4)\mu_{x_4 \to f_c}(x_1)\right) \\ &= \left(\sum_{x_1} f_a(x_1,x_2)\right)\left(\sum_{x_3} f_b(x_2,x_3)\right)\left(\sum_{x_4} f_c(x_2,x_4)\right)\end{aligned}$$



We can now compute marginals at *any* variable, e.g. X_2:

$$
\begin{aligned}
\tilde{p}(x_2) &= \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2) \\
&= \left(\sum_{x_1} f_a(x_1,x_2)\mu_{x_1 \to f_a}(x_1)\right)\left(\sum_{x_3} f_b(x_2,x_3)\mu_{x_3 \to f_b}(x_1)\right)\left(\sum_{x_4} f_c(x_2,x_4)\mu_{x_4 \to f_c}(x_1)\right) \\
&= \left(\sum_{x_1} f_a(x_1,x_2)\right)\left(\sum_{x_3} f_b(x_2,x_3)\right)\left(\sum_{x_4} f_c(x_2,x_4)\right) \\
&= \sum_{x_1}\sum_{x_3}\sum_{x_4} f_a(x_1,x_2)f_b(x_2,x_3)f_c(x_2,x_4)
\end{aligned}
$$



We can now compute marginals at *any* variable, e.g. X$_2$:

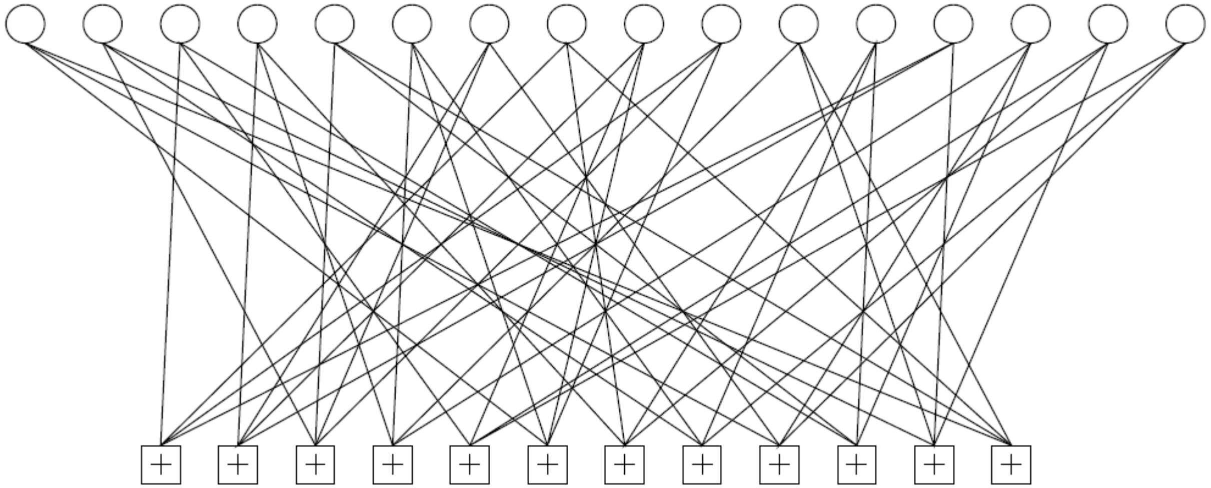
$$
\begin{aligned}
\tilde{p}(x_2) &= \mu_{f_a \to x_2}(x_2)\,\mu_{f_b \to x_2}(x_2)\,\mu_{f_c \to x_2}(x_2) \\
&= \left(\sum_{x_1} f_a(x_1,x_2)\,\mu_{x_1 \to f_a}(x_1)\right)\left(\sum_{x_3} f_b(x_2,x_3)\,\mu_{x_3 \to f_b}(x_1)\right)\left(\sum_{x_4} f_c(x_2,x_4)\,\mu_{x_4 \to f_c}(x_1)\right) \\
&= \left(\sum_{x_1} f_a(x_1,x_2)\right)\left(\sum_{x_3} f_b(x_2,x_3)\right)\left(\sum_{x_4} f_c(x_2,x_4)\right) \\
&= \sum_{x_1}\sum_{x_3}\sum_{x_4} f_a(x_1,x_2)\,f_b(x_2,x_3)\,f_c(x_2,x_4) \\
&= \sum_{x_1}\sum_{x_3}\sum_{x_4} \tilde{p}(\mathbf{x})
\end{aligned}
$$

Outline

- Sum-Product Belief Propagation
- Loopy Belief Propagation
- Variable Elimination
- Junction Tree Algorithm
- Max-Product Belief Propagation

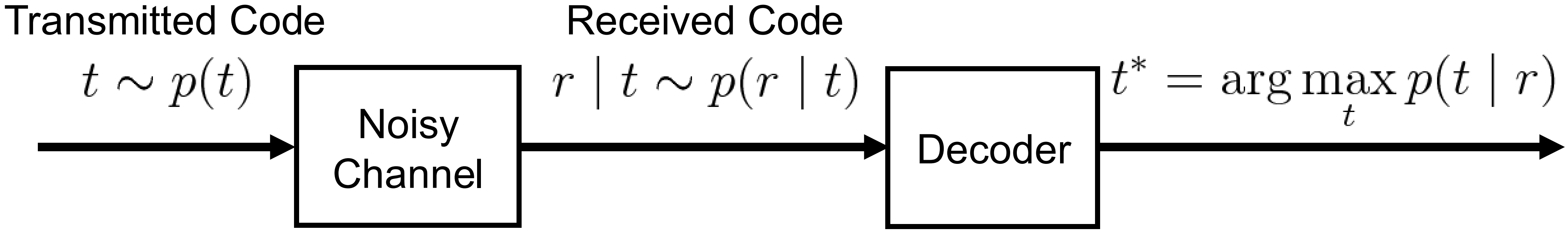
Example: Low Density Parity Check (LDPC) Codes

Factor Graph Representation



Problem Setup

- A code t is transmitted over a noisy
- Received code r is corrupted by noise
- Estimate the most probable code that was sent t* (*maximum a posteriori)*



Transmitted Code

$t \sim p(t)$

Noisy Channel

Received Code

$r \mid t \sim p(r \mid t)$

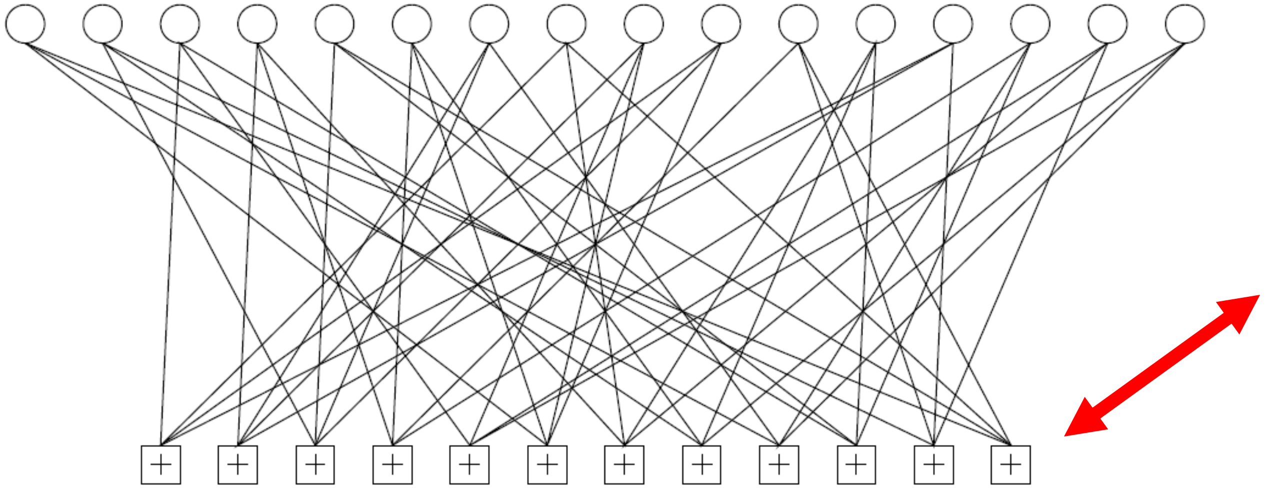
Decoder

$t^* = \arg\max_t p(t \mid r)$

[Source: David MacKay]

Example: Low Density Parity Check (LDPC) Codes

Factor Graph Representation



Sparse Parity Check Matrix

$$\mathbf{H} = \left[\begin{array}{cccccccccccccccc}
 & & 1 & & & & & 1 & & 1 & & & 1 & & & \\
 & & & 1 & & & 1 & & 1 & & & & 1 & & & \\
 & 1 & & & 1 & & 1 & & & 1 & & & & & & \\
 & & & 1 & & 1 & & & & & 1 & & & 1 & & \\
 & & 1 & & & & & & 1 & & & & & & 1 & 1 \\
1 & & & & & 1 & & & 1 & 1 & & & & & & \\
 & & & 1 & & & & 1 & & & & 1 & & & 1 & \\
 & 1 & & & & 1 & & & & & & 1 & & & & 1 \\
1 & & & & & & 1 & & & & & & & 1 & & 1 \\
 & & 1 & & 1 & & & & & & & 1 & & 1 & & \\
 & 1 & & & & & & & & & 1 & & 1 & & 1 & \\
1 & & & & 1 & & & 1 & & & 1 & & & & &
\end{array}\right]$$

- Valid codes have zero parity: $p(t) \propto \mathbb{I}(Ht = 0 \bmod 2)$
- Chanel noise model arbitrary, e.g. flip bits w/ ϵ probability:

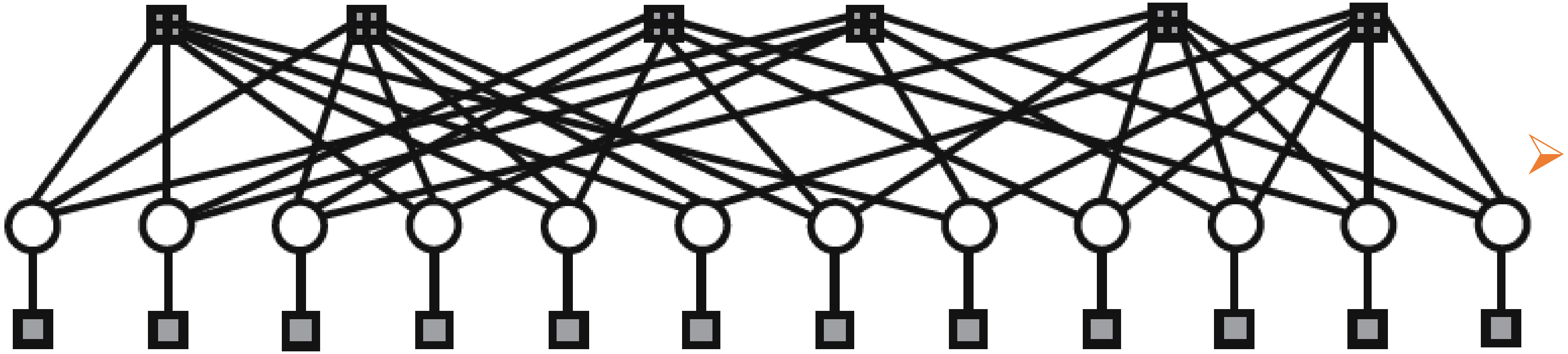
$$p(r \mid t) = \prod_n p(r_n \mid t_n) = \prod_n (1-\epsilon)^{\mathbb{I}(r_n = t_n)} \epsilon^{\mathbb{I}(r_n \neq t_n)}$$

n-th bit →

[Source: David MacKay]

Example: Low Density Parity Check (LDPC) Codes

Parity Check Factors

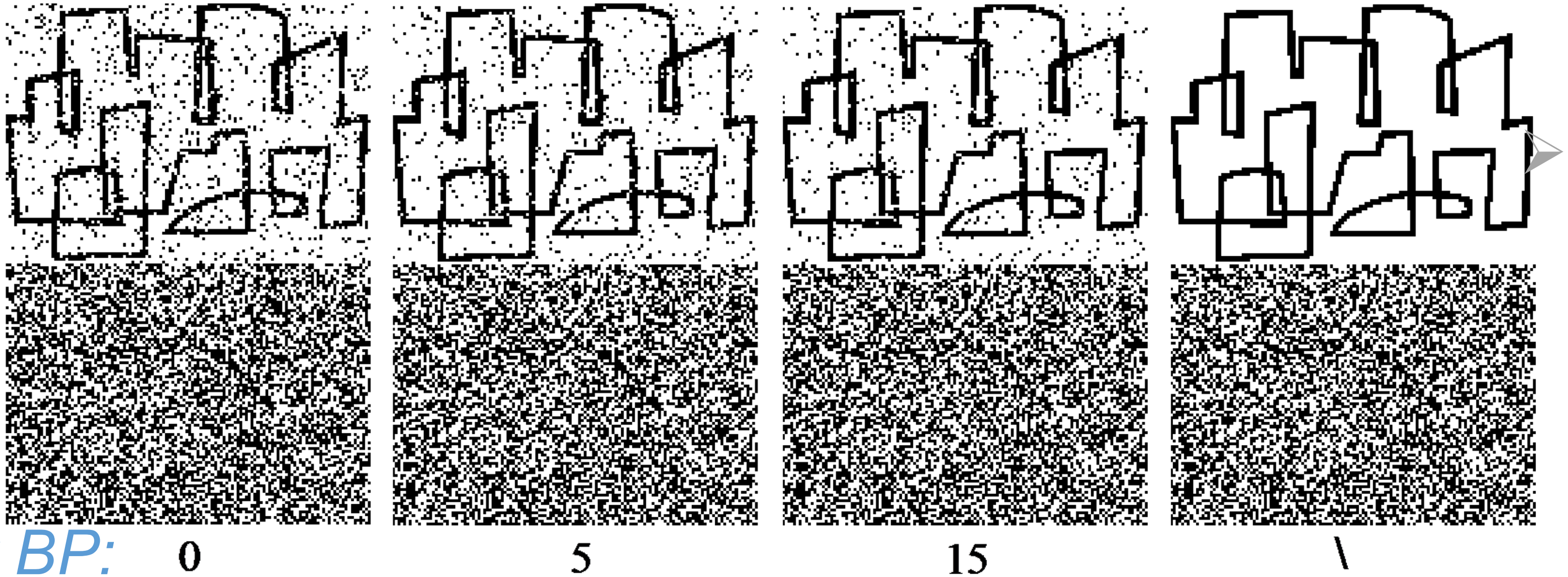


Evidence (observation) Factors

Data Bits

Parity Bits

Loopy BP:



- Each variable node is binary, so $x_s \in \{0, 1\}$
- *Parity check factors* equal 1 if the sum of the connected bits is even, 0 if the sum is odd (invalid codewords are excluded)
- *Unary evidence factors* equal probability that each bit is a 0 or 1, given data. Assumes independent "noise" on each bit.

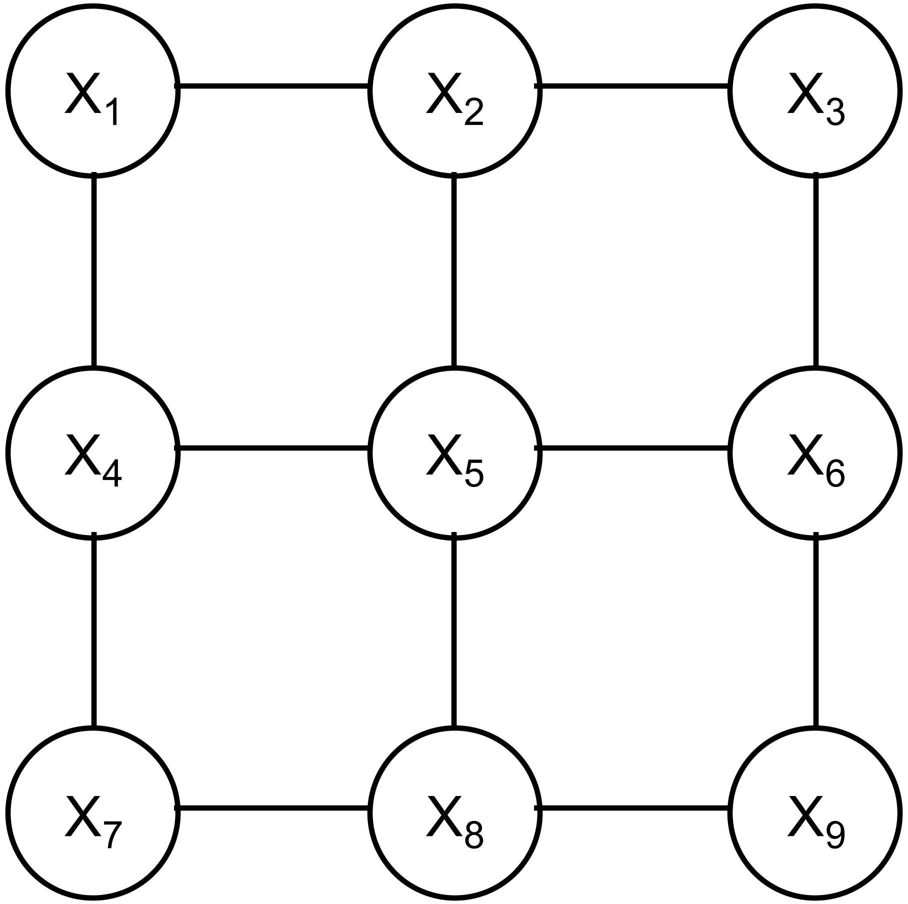
BP for Loopy Graphs

Suppose we have a graph with cycles…

Sum-product BP for tree-structured graphs relies on a leaf-to-root / root-to-leaf sequential update schedule

Graphs with cycles are "loopy" and have no obvious message ordering

Where do we even start? Every node requires initial messages…



BP for Loopy Graphs

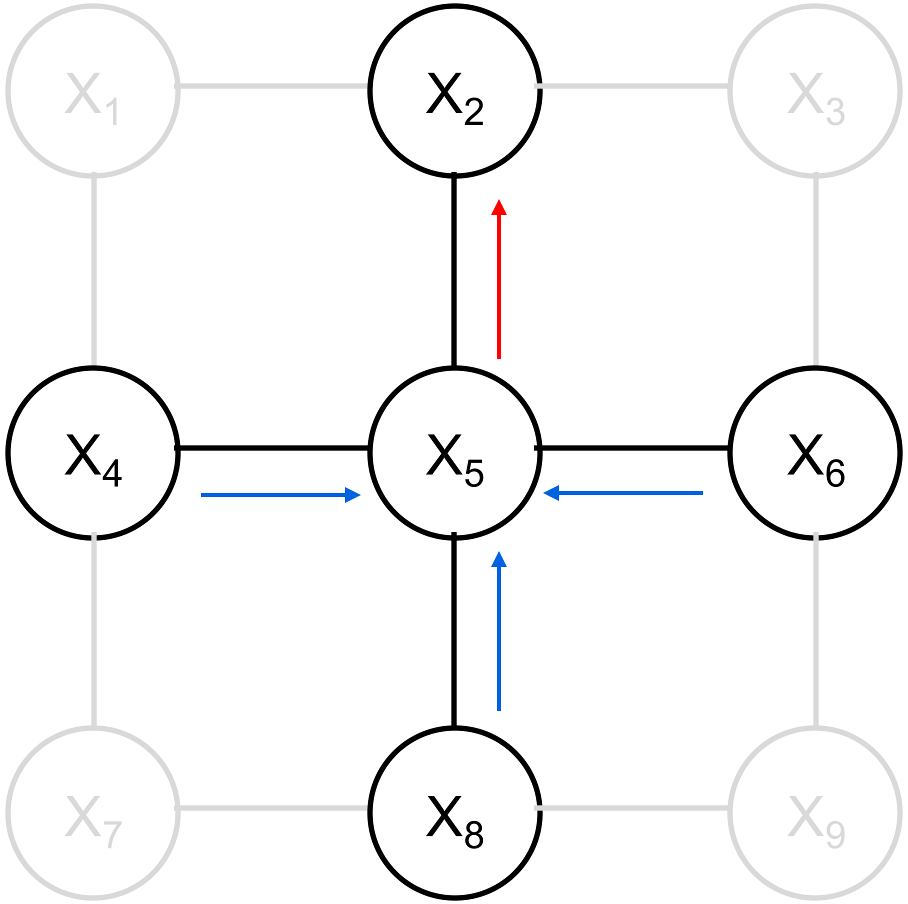
Observe BP message update only depends on Markov Blanket:

$$m_{52}(x_2) = \sum_{x_5} \psi(x_2, x_5) \prod_{k \in \Gamma(5) \backslash 2} m_{k5}(x_5)$$

Where Γ is the set of neighbors:

$$\Gamma(s) = \{t : (s,t) \in \mathcal{E}\}$$

Idea Initialize all messages (somehow) then iteratively update each message until "convergence".



What is convergence? Will this converge? If so, then to what?

Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \text{const.}$

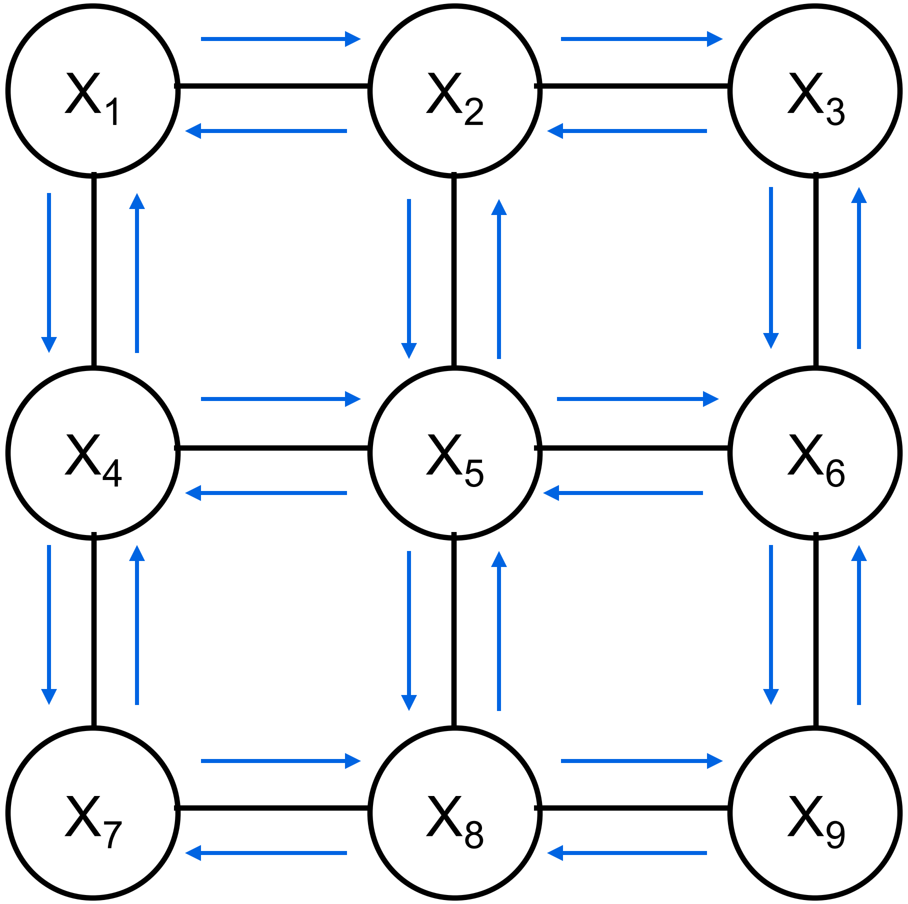
Random: $m^0_{st}(x_t) \sim U([0,1])$

Parallel (Synchronous) Updates

At iteration i update *all messages in parallel* using current messages m^{i-1} from previous iteration:

$$m^i_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m^{i-1}_{ks}(x_s)$$

- Store, both, the *previous* messages (from iteration i-1) and *current* messages (from iteration i)
- Many convergence results assume parallel updates



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m_{st}^{0}(x_t) = \text{const.}$

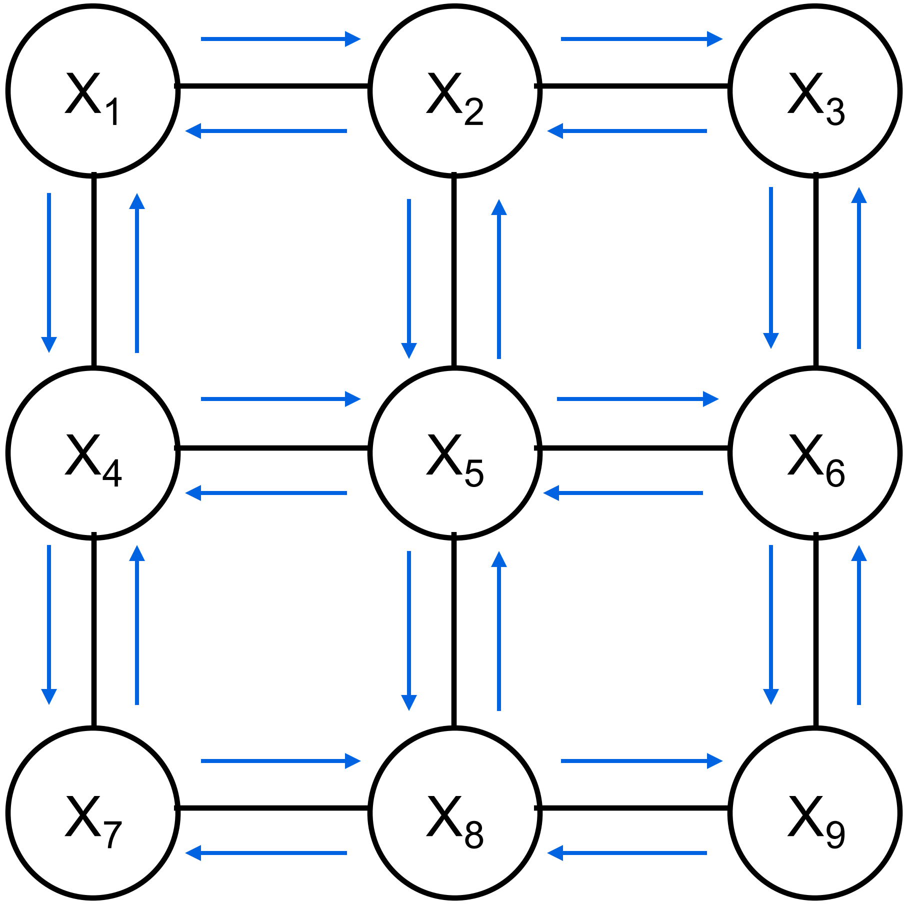
Random: $m_{st}^{0}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \text{const.}$

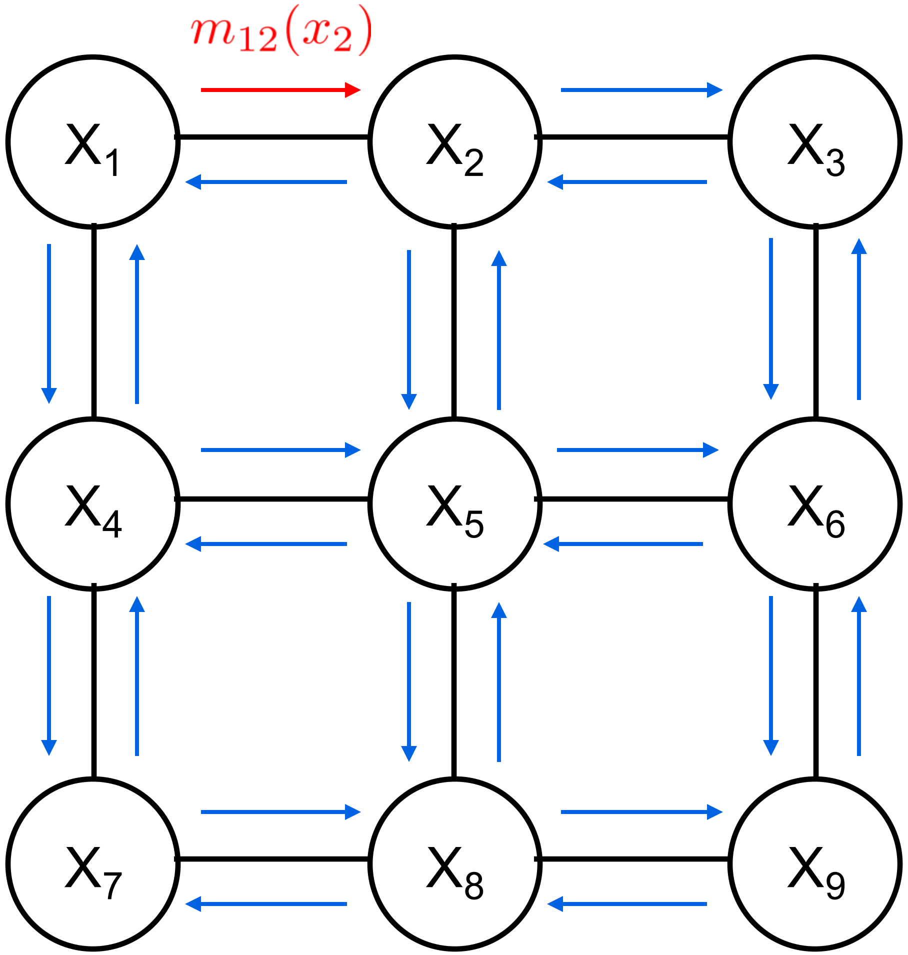
Random: $m^0_{st}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \text{const.}$

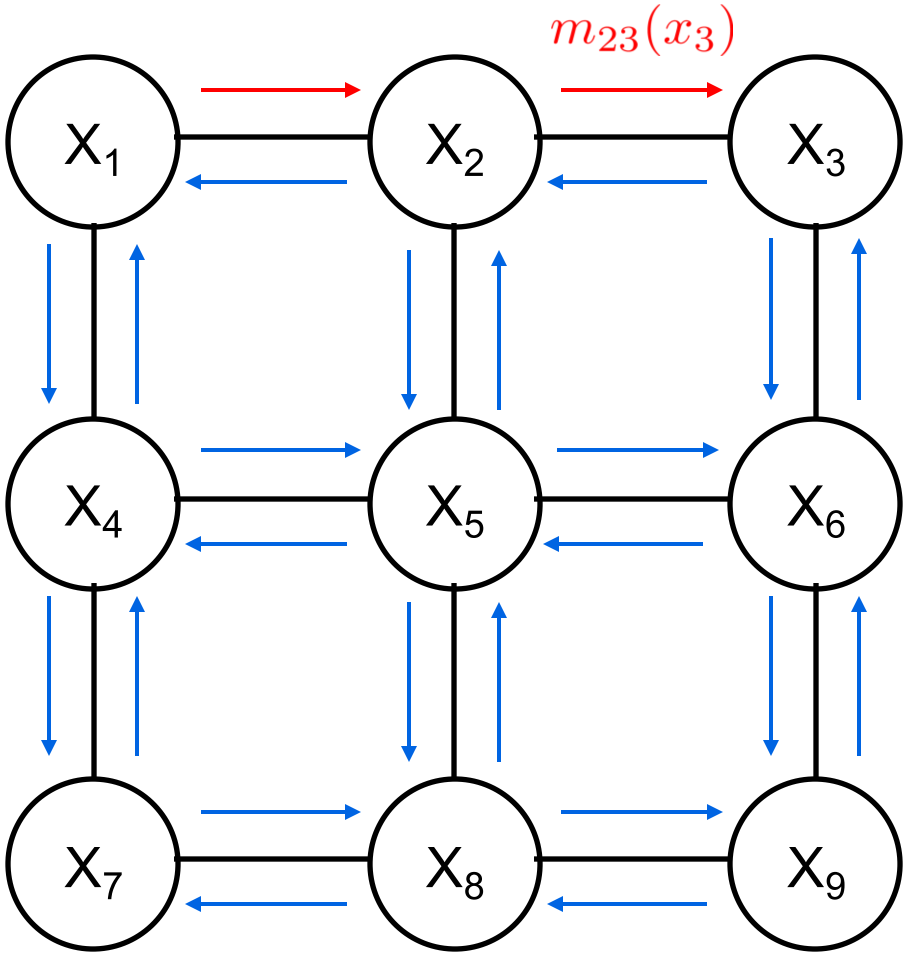
Random: $m^0_{st}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m_{st}^{0}(x_t) = \text{const.}$

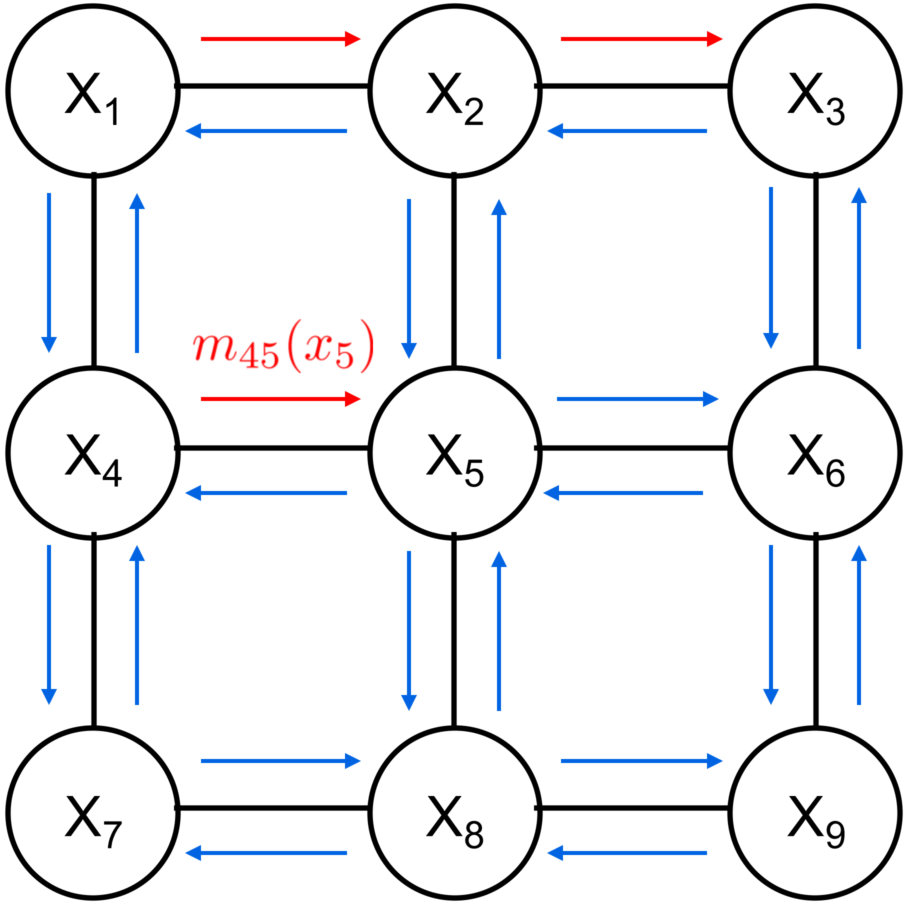
Random: $m_{st}^{0}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \text{const.}$

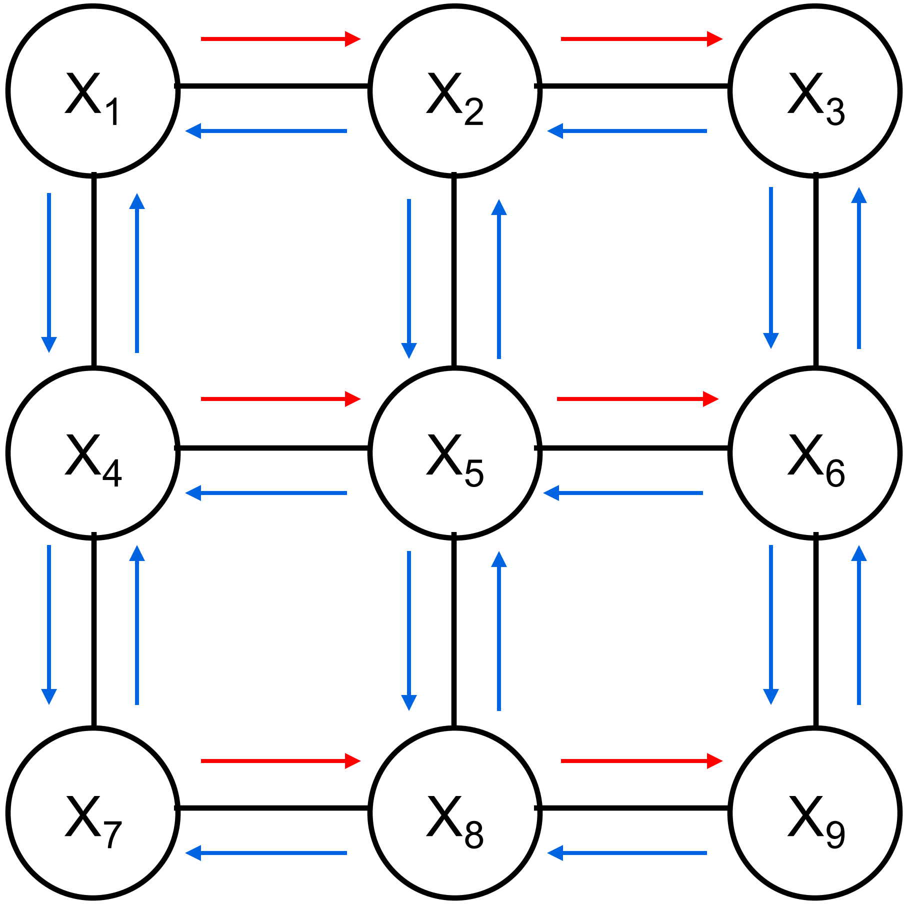
Random: $m^0_{st}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Notice that each row can be computed in parallel

Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m_{st}^{0}(x_t) = \text{const.}$

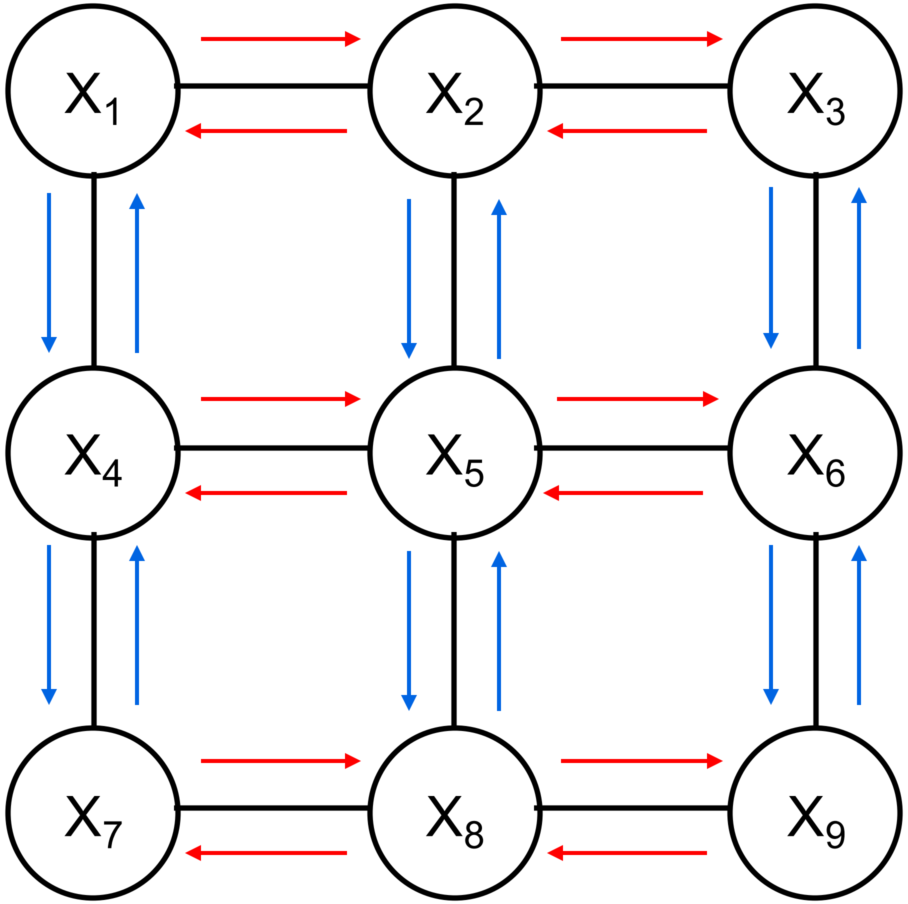
Random: $m_{st}^{0}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \backslash t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Both directions are independent just like in forward-backward algorithm

Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \mathrm{const.}$

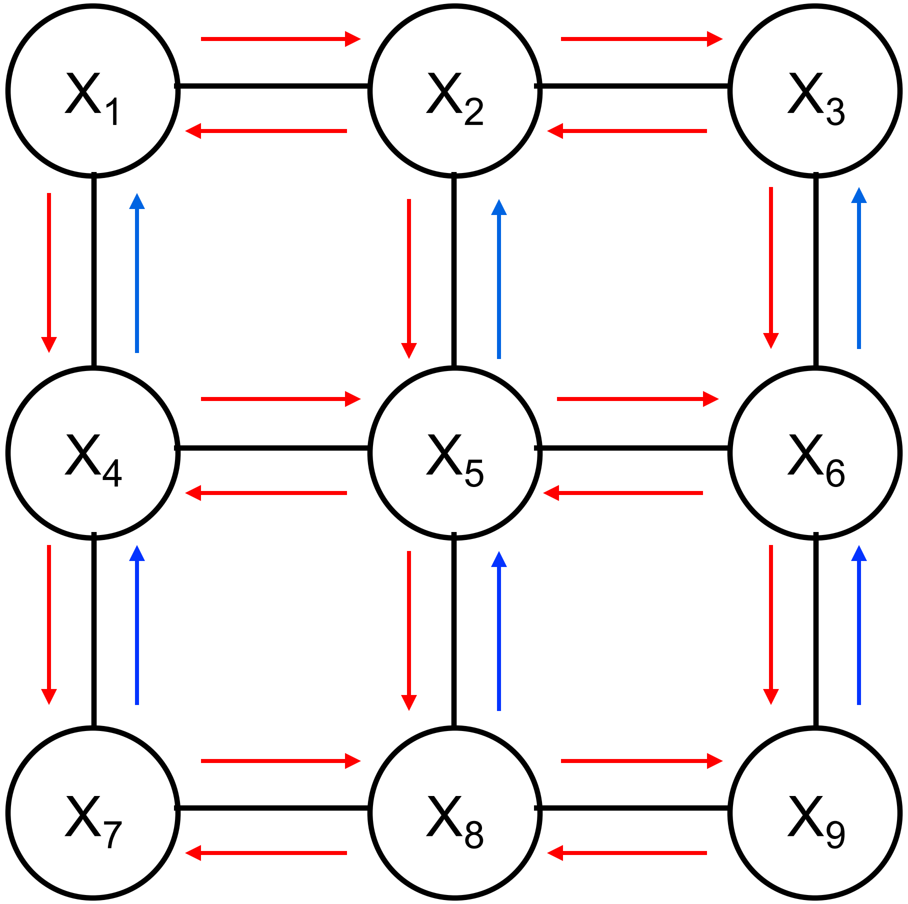
Random: $m^0_{st}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Loopy Belief Propagation (sum-product)

Initialize Messages

Constant: $m^0_{st}(x_t) = \text{const.}$

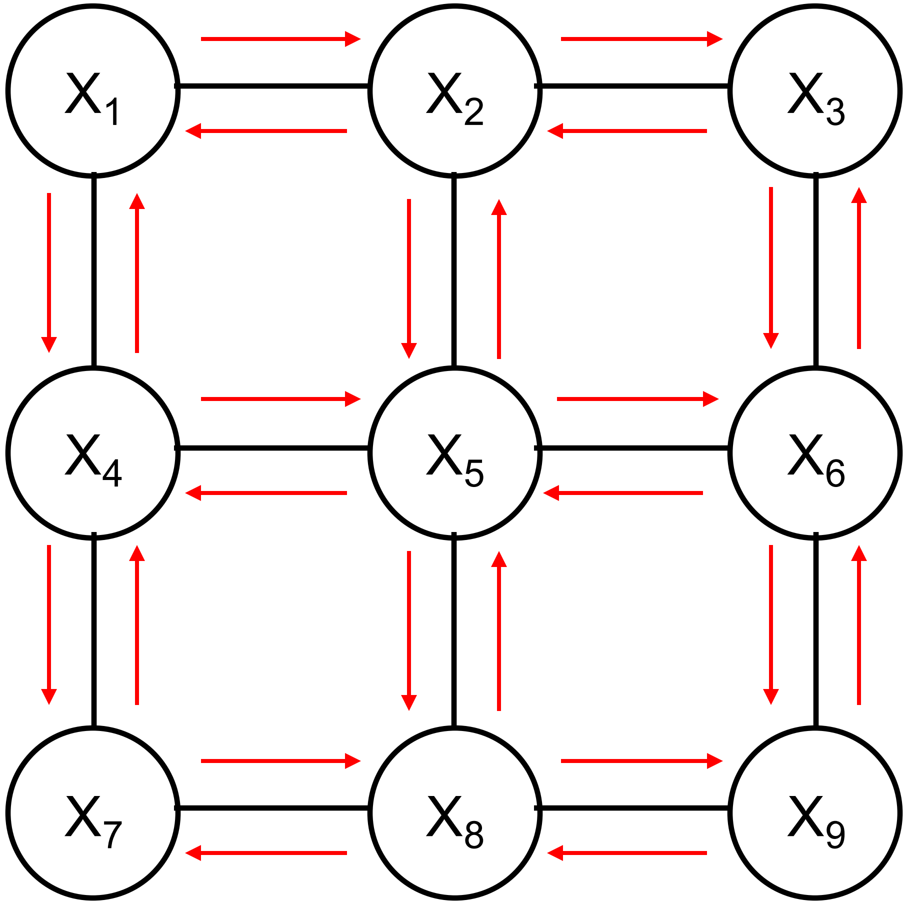
Random: $m^0_{st}(x_t) \sim U([0,1])$

Asynchronous (Sequential) Updates

Choose an ordering of nodes and update using the latest available messages:

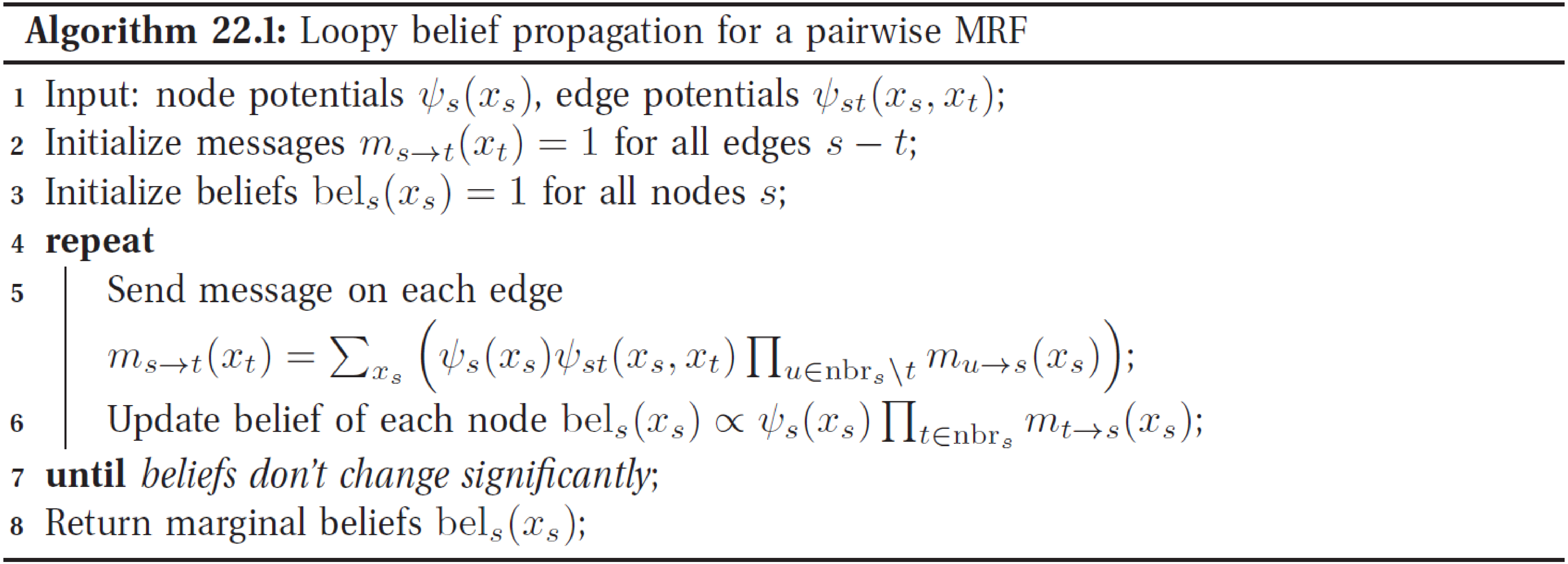
$$m_{st}(x_t) = \sum_{x_s} \psi_{st}(x_s, x_t) \prod_{k \in \Gamma(s) \setminus t} m_{ks}(x_s)$$

- Simplifies updates since only need to keep track of one copy of messages
- Makes parallel processing trickier



Upwards / downwards directions can also be done in parallel (holding rows fixed)

Pseudocode from Murphy’s Textbook



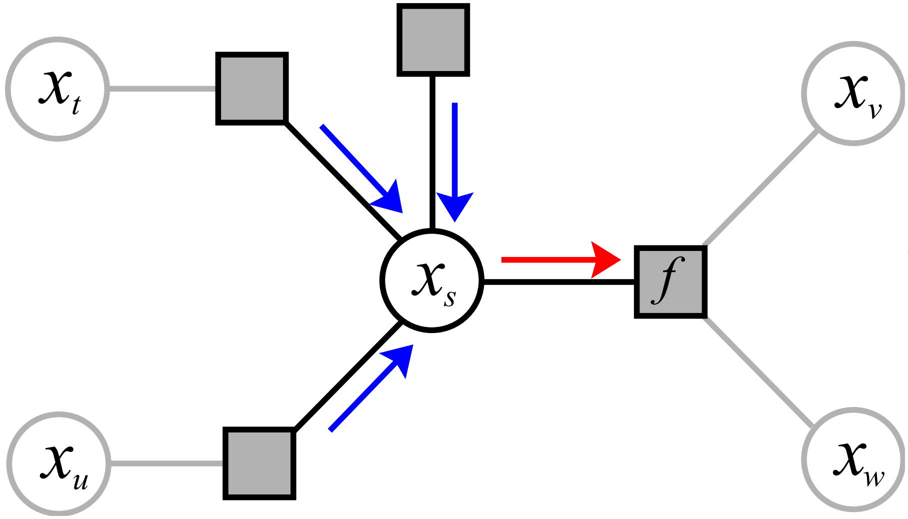
Algorithm 22.1: Loopy belief propagation for a pairwise MRF

```
Input: node potentials $\psi_s(x_s)$, edge potentials $\psi_{st}(x_s, x_t)$;
Initialize messages $m_{s \to t}(x_t) = 1$ for all edges $s - t$;
Initialize beliefs $\text{bel}_s(x_s) = 1$ for all nodes $s$;
repeat
    Send message on each edge
    $m_{s \to t}(x_t) = \sum_{x_s} \left( \psi_s(x_s) \psi_{st}(x_s, x_t) \prod_{u \in \text{nbr}_s \setminus t} m_{u \to s}(x_s) \right)$;
    Update belief of each node $\text{bel}_s(x_s) \propto \psi_s(x_s) \prod_{t \in \text{nbr}_s} m_{t \to s}(x_s)$;
until beliefs don’t change significantly;
Return marginal beliefs $\text{bel}_s(x_s)$;
```

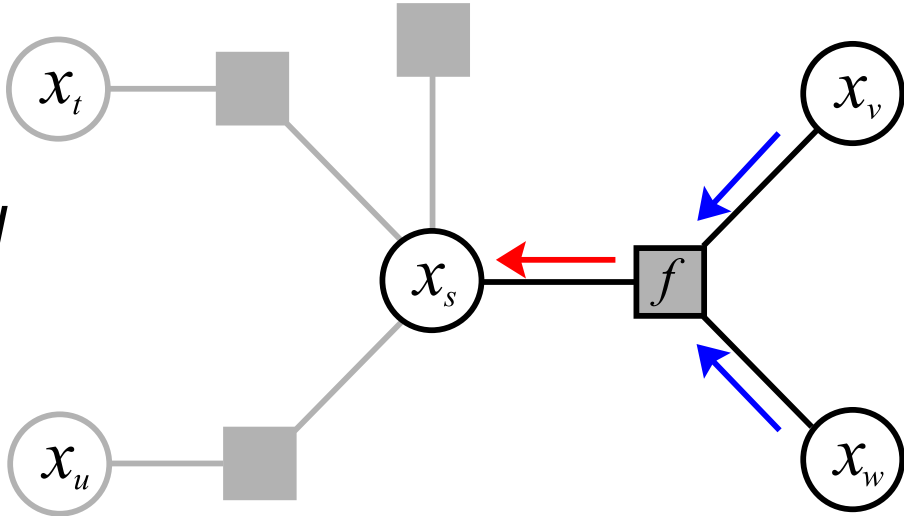
Loopy BP on Factor Graphs

Set of *neighbors* of node *s*: $\Gamma(s) = \{f \in \mathcal{F} \mid s \in f\}$

[Source: Erik Sudderth]



Loopy BP:
Message updates can be iteratively computed on graphs with cycles.
But marginals not guaranteed correct!



$$\bar{m}_{sf}(x_s) = \prod_{g \in \Gamma(s) \setminus f} m_{gs}(x_s) \propto \frac{p_s(x_s)}{m_{fs}(x_s)}$$

$$m_{fs}(x_s) = \sum_{x_{f \setminus s}} \psi_f(x_f) \prod_{t \in f \setminus s} \bar{m}_{tf}(x_t)$$

Marginal Distribution of Each Variable:

$$p_s(x_s) \propto \prod_{f \in \Gamma(s)} m_{fs}(x_s)$$

Marginal Distribution of Each Factor:
Clique of variables linked by factor.

$$p_f(x_f) \propto \psi_f(x_f) \prod_{s \in f} \bar{m}_{sf}(x_s)$$

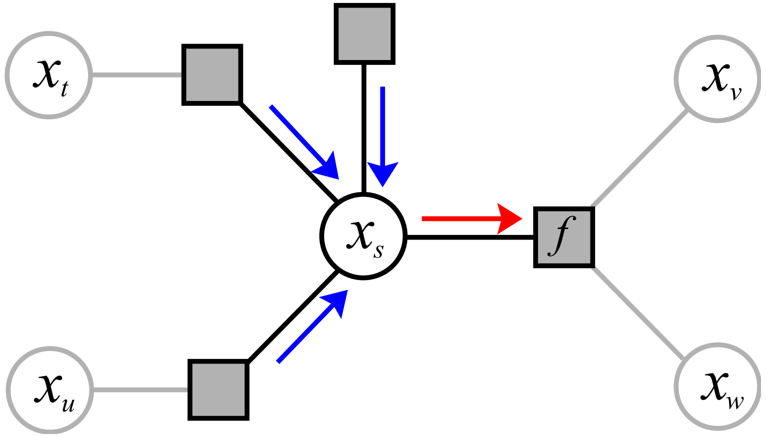
Numerical Stability

Product over messages is numerically unstable…

$$\bar{m}_{sf}(x_s) = \prod_{g \in \Gamma(s) \setminus f} m_{gs}(x_s) \propto \frac{p_s(x_s)}{m_{fs}(x_s)}$$

Product of small values tends to underflow

Vector of small values



1. Do the product as a summation in log-domain:

$$\log \bar{m}_{sf}(x_s) = \sum_g \log m_{gs}(x_s)$$

2. Subtract the maximum value (this makes new maximum zero):

$$\alpha = \max_{x_s} \log \bar{m}_{sf}(x_s) \qquad \log \bar{m}_{sf}(x_s) = \log \bar{m}_{sf}(x_s) - \alpha$$

3. Exponentiate (optionally normalize):

$$\bar{m}_{sf}(x_s) = \exp\left(\log \bar{m}_{sf}(x_s)\right) \div \left(\sum_{x_s} \exp\left(\log \bar{m}_{sf}(x_s)\right) \right)$$

Loopy BP Convergence

Loopy BP works well empirically, but there are no guarantees:

- Not guaranteed to converge in general graphs
- BP marginal *beliefs* are **approximations**
- Empirically, when LBP converges it does so quickly and with good approximations

Convergence based on change in messages / marginal approximations:

$$\rho(m^{\text{old}}, m^{\text{current}}) < \epsilon \qquad \text{or} \qquad \rho(\text{bel}^{\text{old}}, \text{bel}^{\text{current}}) < \epsilon$$

Typical convergence measures are:

Max change: $\rho(m^{\text{old}}, m^{\text{current}}) = \max \left\{ |m^{\text{old}} - m^{\text{current}}| \right\}$

Total change: $\rho(m^{\text{old}}, m^{\text{current}}) = \sum |m^{\text{old}} - m^{\text{current}}|$

Loopy BP Convergence

What can we do to improve convergence in a given model?

Message damping takes a *partial update* of messages each iteration,

$$m^{\text{new}} = (1-\alpha)m^{\text{old}} + \alpha m^{\text{tmp}}$$

for damping factor $\alpha \in (0,1]$, e.g. $\alpha = 1$ is standard update

Message scheduling

- Asynchronous updates tend to converge faster than synchronous
- Well-known Gauss-Seidel method does this in round-robin fashion (Bertsekas 97)
- Message update ordering also impacts convergence (e.g. disproportionate impact of nodes 2 & 3 in previous example)

Example: Loopy BP

Convergence depends largely on the existence of many small cycles

Example Ising model of ferromagnetism via atomic *spins:*

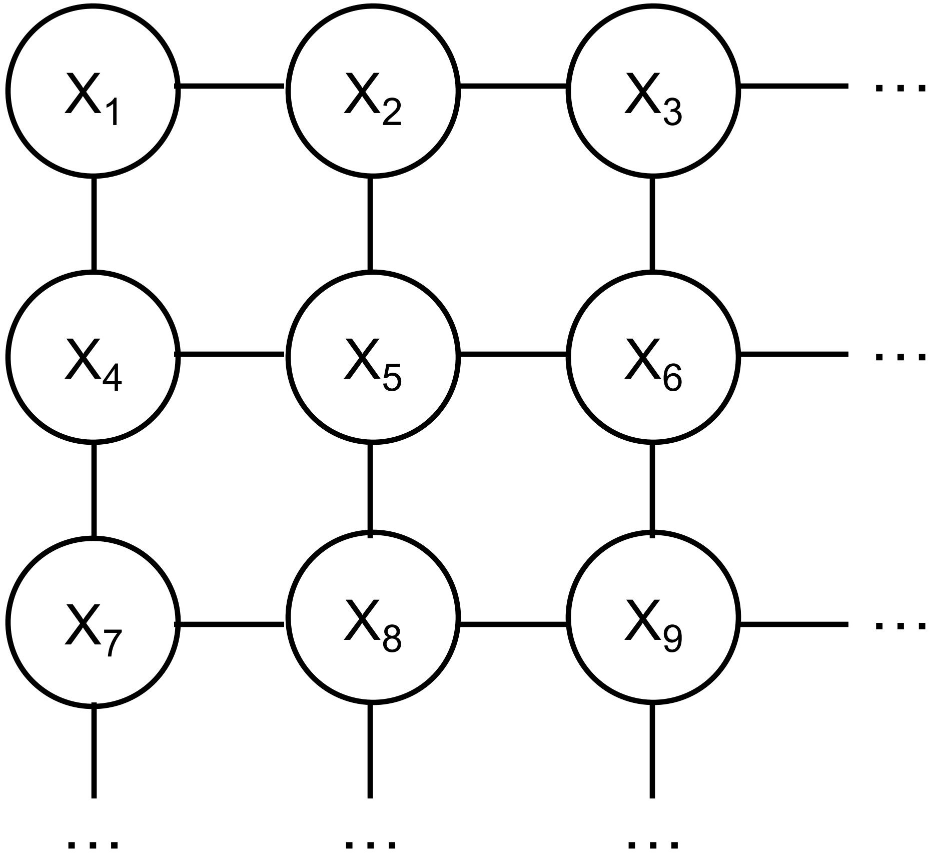
Binary *spin* variables: $x_i \in \{0, 1\}$

Interaction strength:

$$\psi_{ij} = \begin{pmatrix} \exp(J_{ij}) & \exp(-J_{ij}) \\ \exp(-J_{ij}) & \exp(J_{ij}) \end{pmatrix}$$

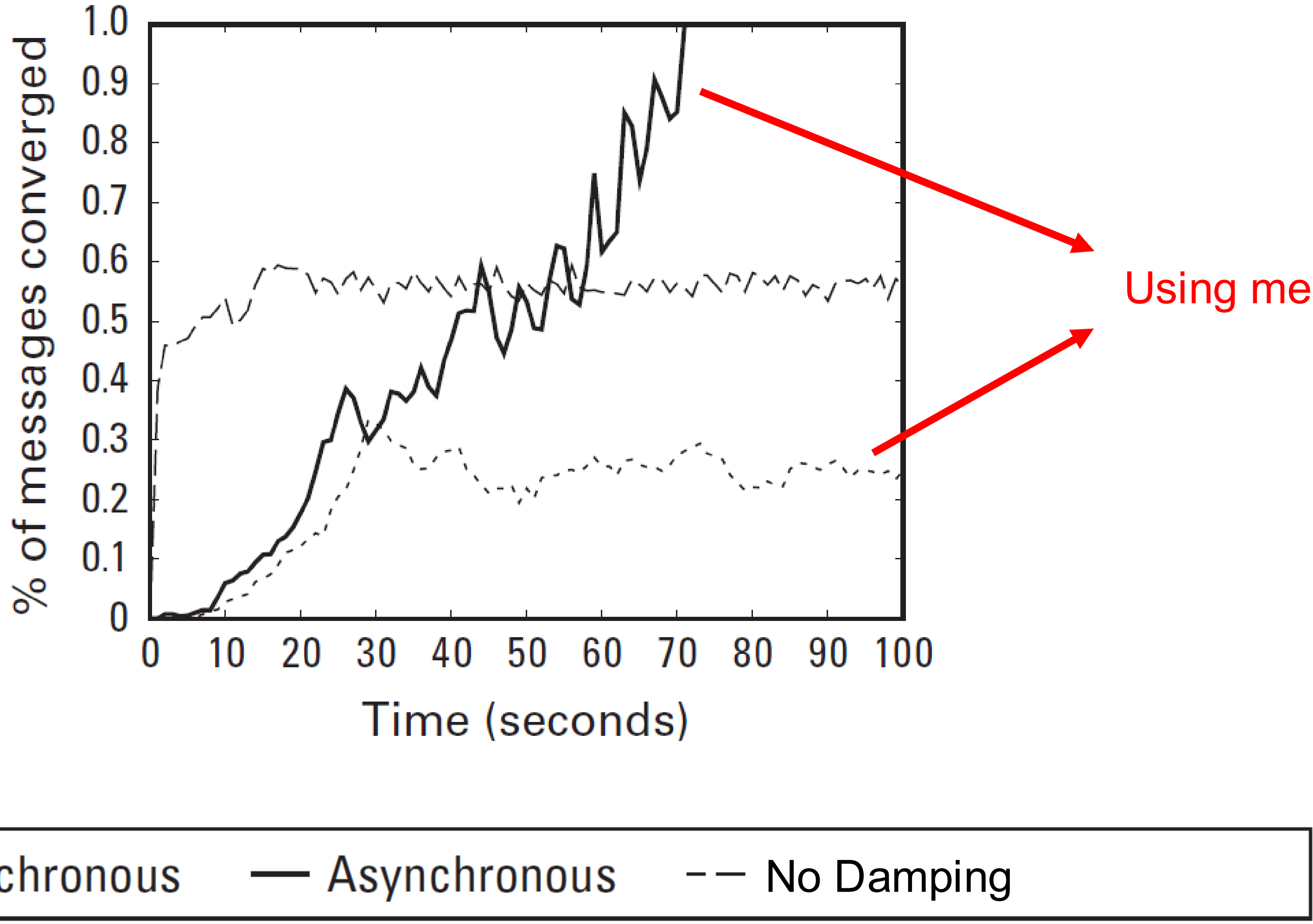
Field strength:

$$\psi_i = (\exp(h_i); \grave{\exp}(-h_i))$$



Example: Loopy BP

11x11 Ising model with random parameters

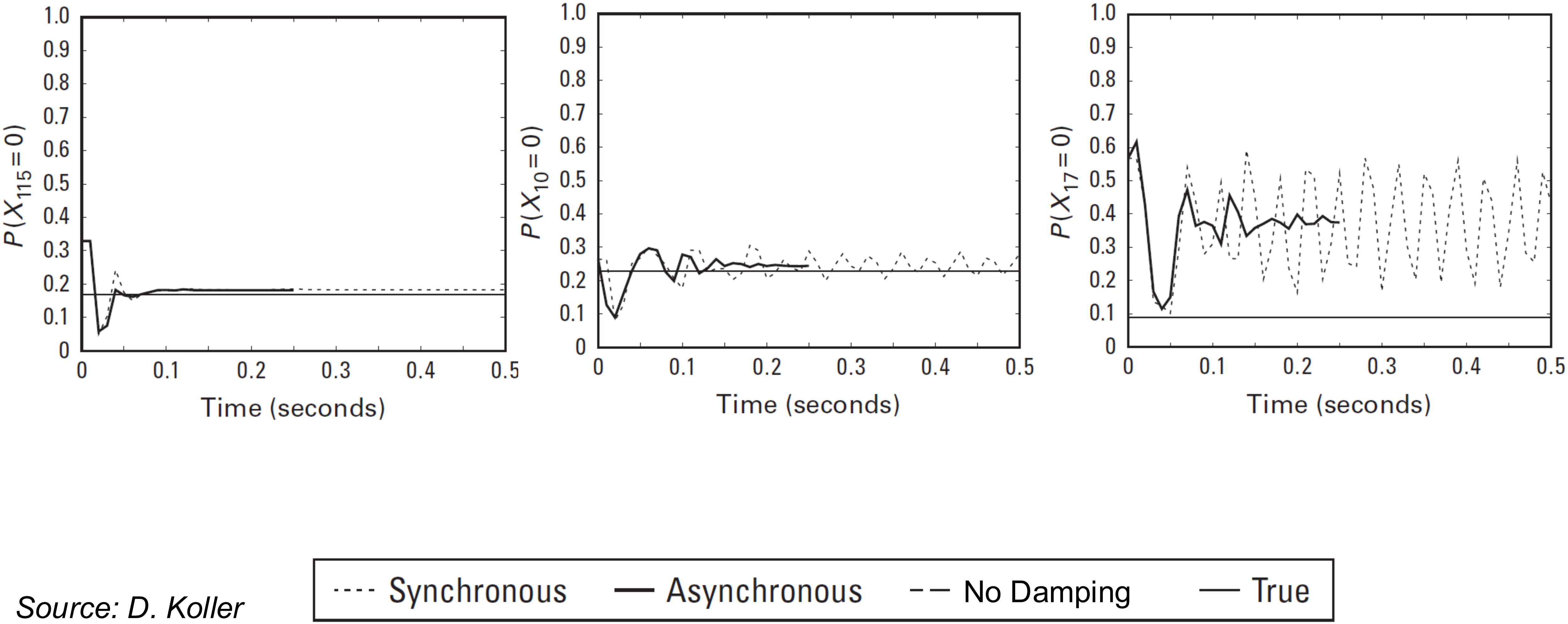


Using message damping

Source: D. Koller

Example: Loopy BP

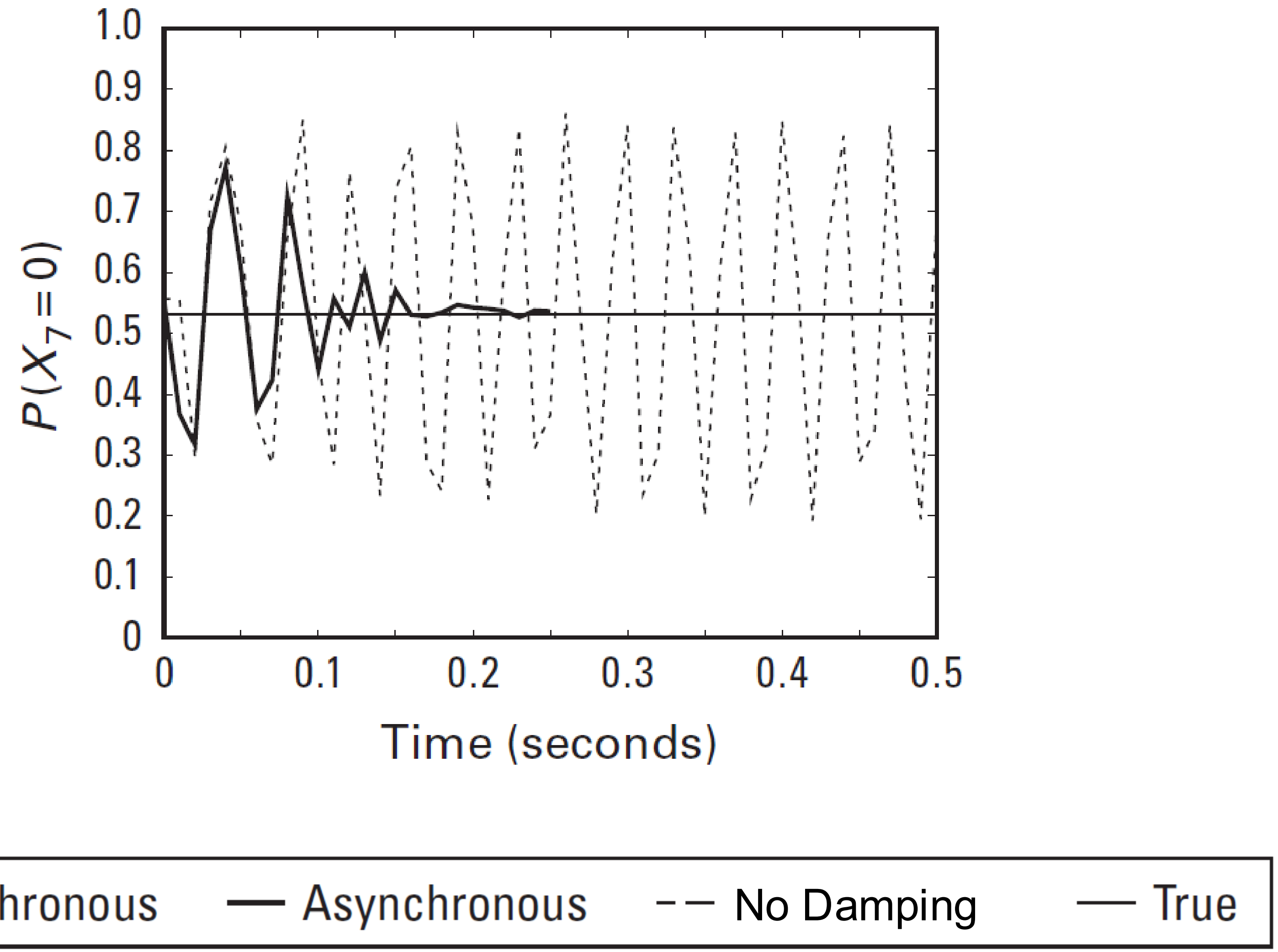
Convergence of beliefs in 3 selected nodes



Source: D. Koller

Example: Loopy BP

Oscillation in limit cycles is a typical failure mode of BP convergence



Source: D. Koller

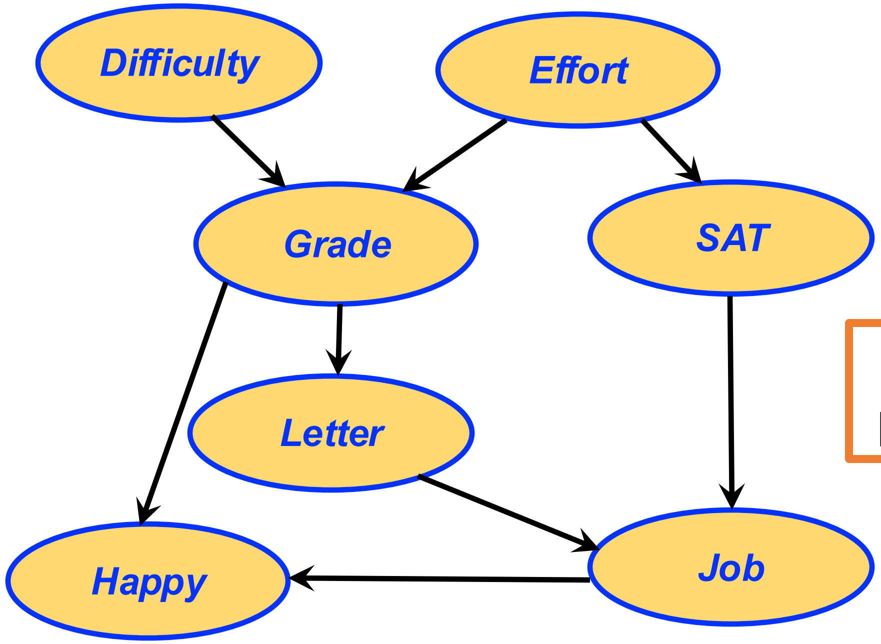
Loopy BP Summary

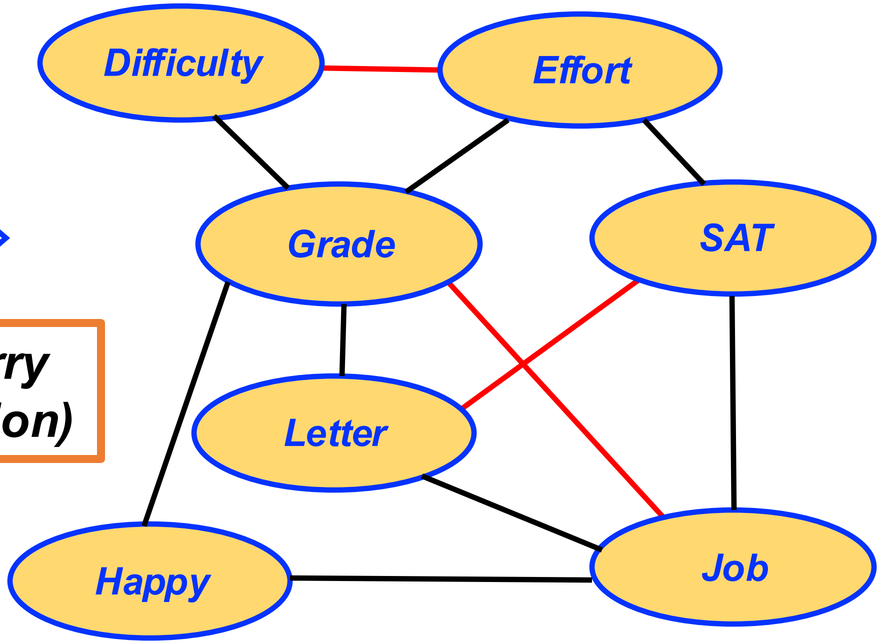
- BP updates only depend on tree-structured Markov blanket
- **Approximate** BP inference in loopy graphs by iterating standard message updates until convergence (fixed point)
- No guarantees, but works well empirically in many instances
- Some techniques to improve convergence
  - Message damping
  - Asynchronous message update schedules

Outline

- Sum-Product Belief Propagation
- Loopy Belief Propagation
- Variable Elimination
- Junction Tree Algorithm
- Max-Product Belief Propagation

Bayes Net → MRF





Added edges *marry* parents (*moralization*)

$$P(\cdot) = P(D)P(E)P(G \mid D, E)P(S \mid E)P(L \mid G)P(J \mid S, L)P(H \mid J, G)$$

Drop local normalization

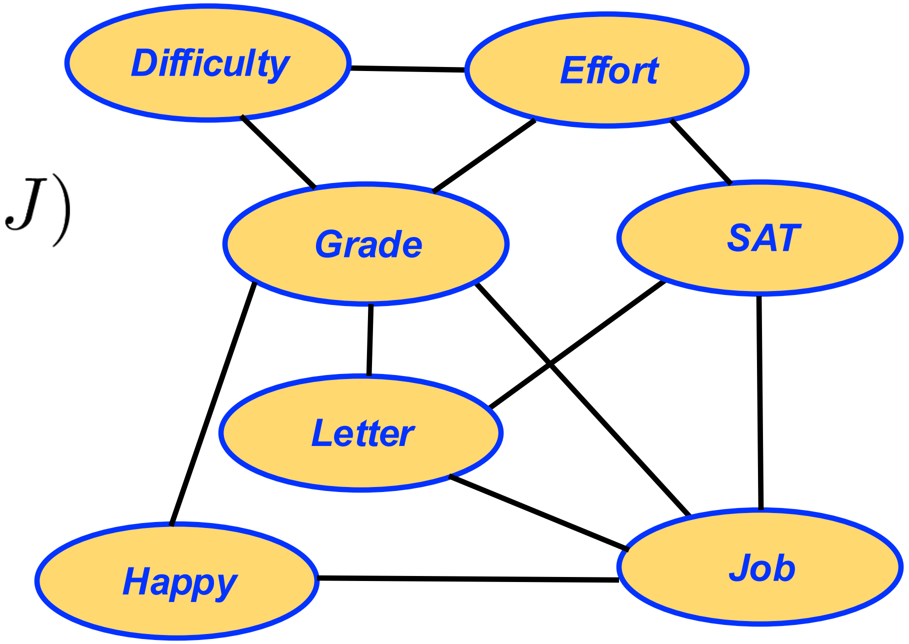
$$P(\cdot) \propto \psi(D)\psi(E)\psi(G, D, E)\psi(S, E)\psi(L, G)\psi(J, S, L)\psi(H, J, G)$$

Variable Elimination Algorithm

What is the probability of getting a job?

$$P(J) = \sum_d \sum_e \sum_h \sum_g \sum_s \sum_l P(d, e, h, g, s, l, J)$$

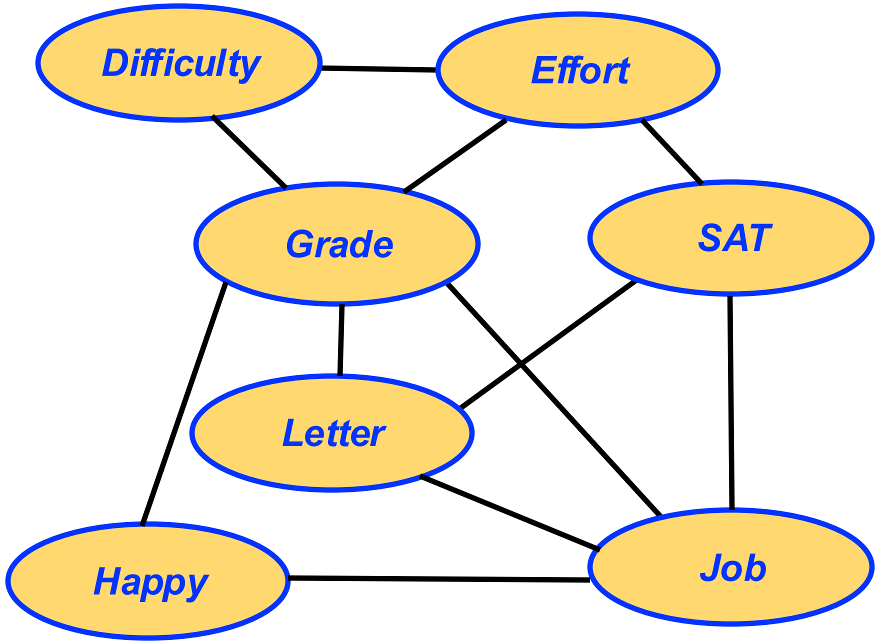
Iteratively eliminate nuisance variables…



$$P(D, E, H, G, S, L, J) \propto \psi(D)\psi(E)\psi(G, D, E)$$
$$\psi(S, E)\psi(L, G)\psi(J, S, L)\psi(H, J, G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L



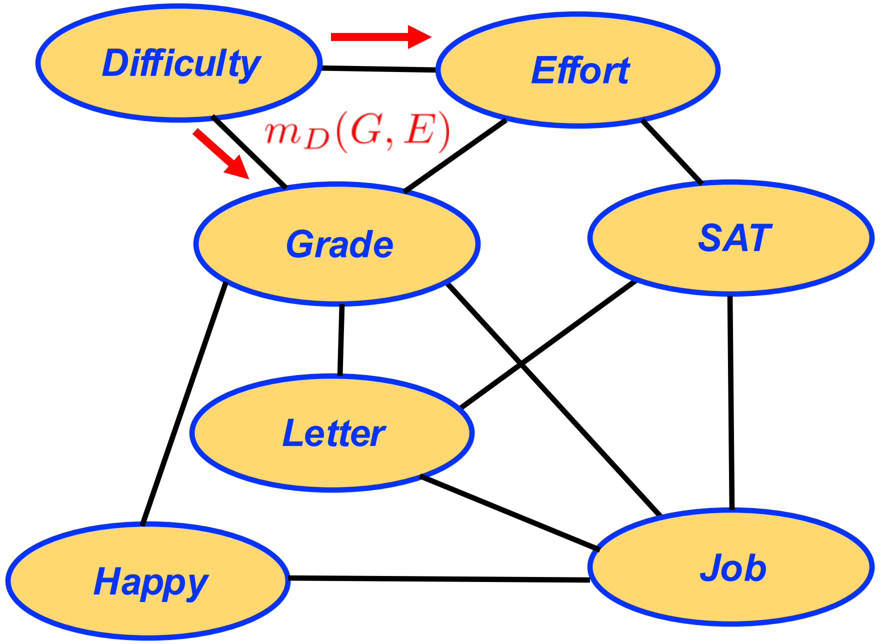
$$P(D,E,H,G,S,L,J) \propto \psi(D)\psi(E)\psi(G,D,E)$$
$$\psi(S,E)\psi(L,G)\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$



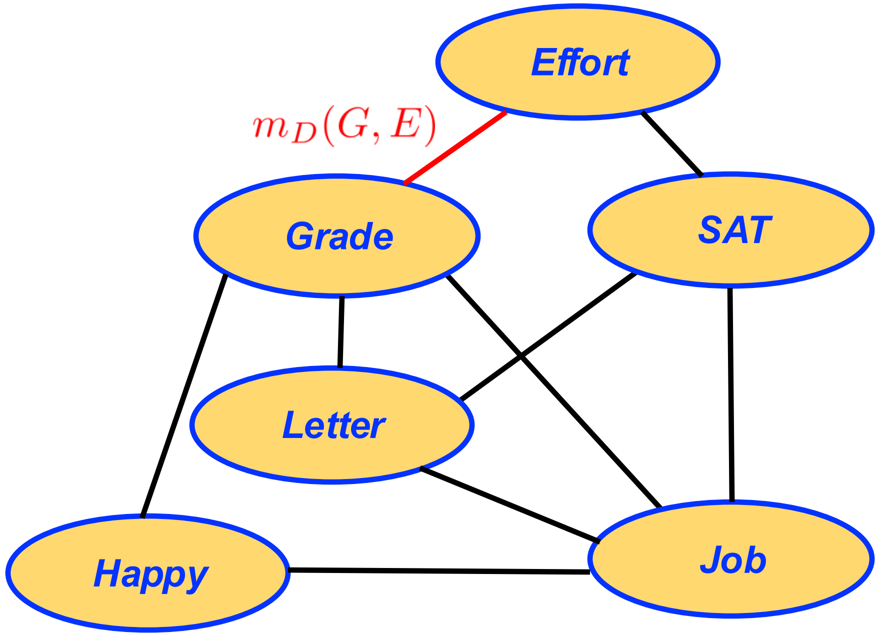
$$P(D,E,H,G,S,L,J) \propto \psi(D)\psi(E)\psi(G,D,E)$$
$$\psi(S,E)\psi(L,G)\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$



$$P(E,H,G,S,L,J) \propto m_D(G,E)\psi(E)$$
$$\psi(S,E)\psi(L,G)\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

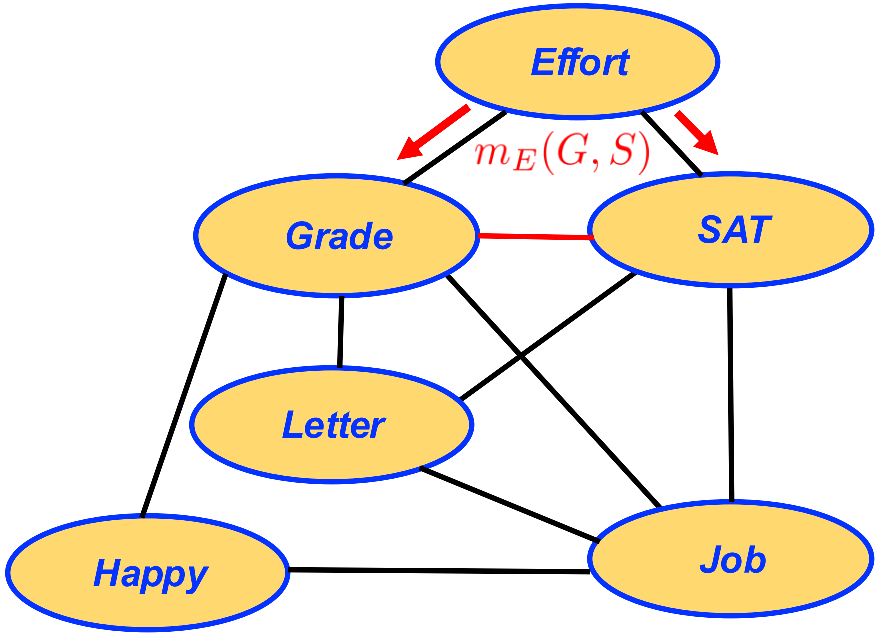
Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$



$$P(E,H,G,S,L,J) \propto m_D(G,E)\psi(E)$$
$$\psi(S,E)\psi(L,G)\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

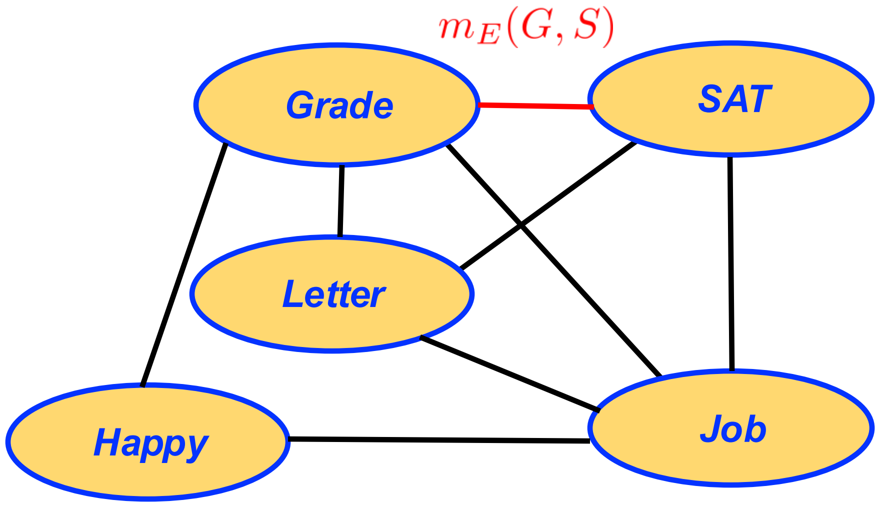
$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$



$$P(H,G,S,L,J) \propto m_E(G,S)\psi(L,G)$$
$$\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

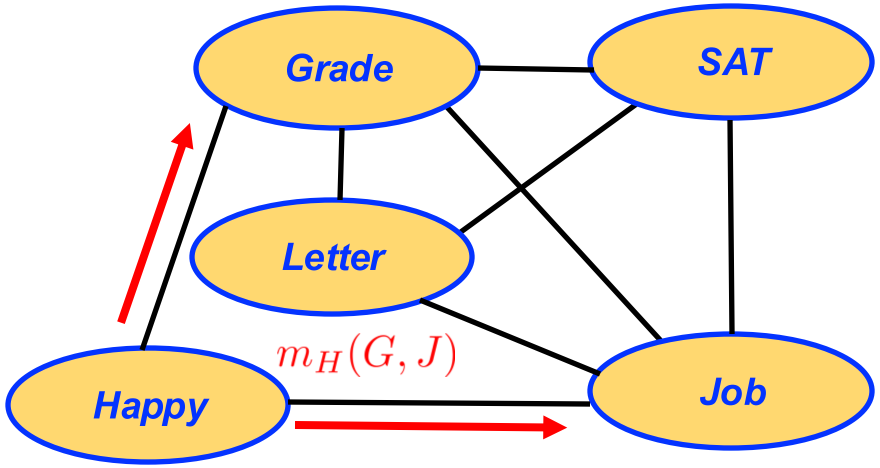
$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$



$$P(H,G,S,L,J) \propto m_E(G,S)\psi(L,G)$$
$$\psi(J,S,L)\psi(H,J,G)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

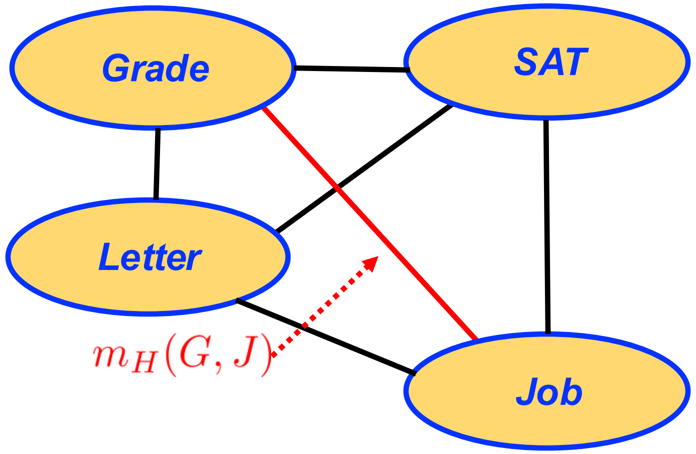
$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$



$$P(G,S,L,J) \propto m_H(G,J) m_E(G,S)\psi(L,G)$$
$$\psi(J,S,L)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

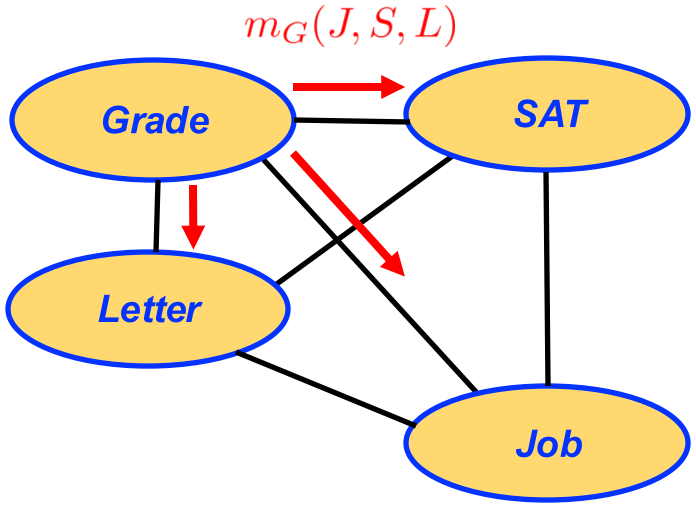
Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$

Eliminate **G** : $m_G(J,S,L) = \sum_g m_H(g,J)m_E(g,S)\psi(L,g)$



$$P(G,S,L,J) \propto m_H(G,J)m_E(G,S)\psi(L,G)$$
$$\psi(J,S,L)$$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

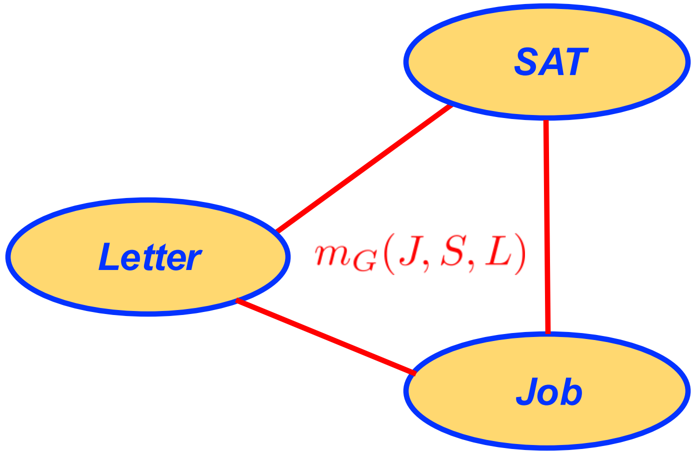
Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$

Eliminate **G** : $m_G(J,S,L) = \sum_g m_H(g,J)m_E(g,S)\psi(L,g)$



$P(S,L,J) \propto m_G(J,S,L)\psi(J,S,L)$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

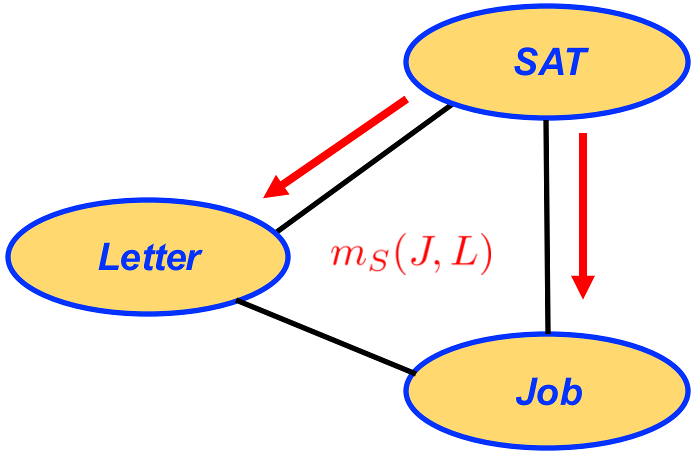
$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$

Eliminate **G** : $m_G(J,S,L) = \sum_g m_H(g,J)m_E(g,S)\psi(L,g)$

Eliminate **S** : $m_S(J,L) = \sum_s m_G(J,s,L)\psi(J,s,L)$



$P(S,L,J) \propto m_G(J,S,L)\psi(J,S,L)$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G,E) = \sum_d \psi(d)\psi(d,G,E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G,S) = \sum_e m_D(G,e)\psi(e)\psi(S,e)$$

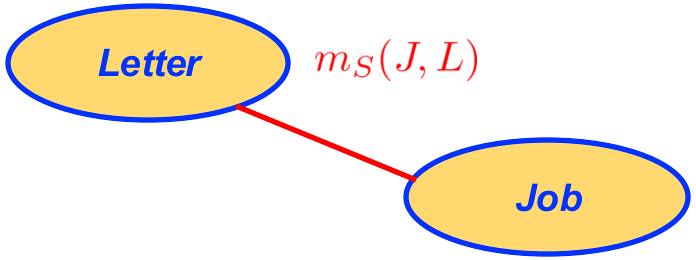
Eliminate **H** (compute message H→(G,J)):

$$m_H(G,J) = \sum_h \psi(h,J,G)$$

Eliminate **G** : $m_G(J,S,L) = \sum_g m_H(g,J)m_E(g,S)\psi(L,g)$

Eliminate **S** : $m_S(J,L) = \sum_s m_G(J,s,L)\psi(J,s,L)$

Eliminate **L** : $m_L(J) = \sum_l m_S(J,l)$



$P(L,J) \propto m_S(J,L)$

Variable Elimination Algorithm

Choose elimination ordering: D, E, H, G, S, L

Eliminate **D** (compute message D→(G,E)):

$$m_D(G, E) = \sum_d \psi(d)\psi(d, G, E)$$

Eliminate **E** (compute message E→(G,S)):

$$m_E(G, S) = \sum_e m_D(G, e)\psi(e)\psi(S, e)$$

Eliminate **H** (compute message H→(G,J)):

$$m_H(G, J) = \sum_h \psi(h, J, G)$$

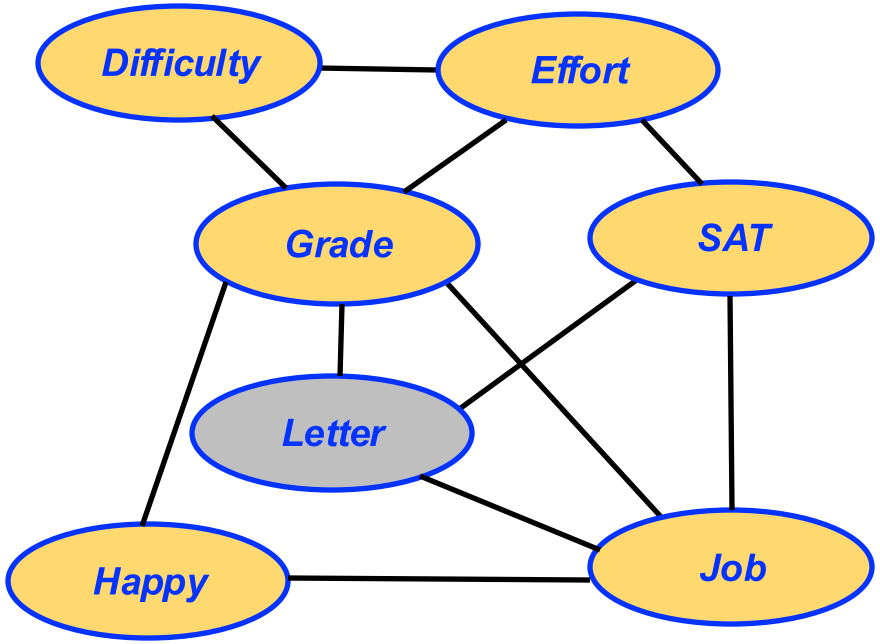
Eliminate **G** : $m_G(J, S, L) = \sum_g m_H(g, J) m_E(g, S)\psi(L, g)$

Eliminate **S** : $m_S(J, L) = \sum_s m_G(J, s, L)\psi(J, s, L)$

Eliminate **L** : $m_L(J) = \sum_l m_S(J, l) \propto P(J)$

$P(J) \propto m_l(J)$

Accounting for Evidence



What if we observe a node (e.g. Letter=l)?

$$P(J \mid L = l) = \frac{P(J, L = l)}{P(L = l)}$$

Step 1: *Clamp* $L = l$ in any factor with L:

$$P(D, E, H, G, S, L = l, J) \propto \psi(D)\psi(E)\psi(G, D, E)$$
$$\psi(S, E)\psi(L = l, G)\psi(J, S, L = l)\psi(H, J, G)$$

Just treat these as new factors, since we don’t care about normalizer:

$\psi'(G) = \psi(L = l, G)$ and $\psi'(J, S) = \psi(J, S, L = l)$

Step 2: Remove L from elimination ordering

Computational Complexity

Main Points:

- Worst-case complexity of variable elimination is **exponential** in the number of latent variables

- Complexity is dependent on chosen elimination order

Computational Complexity

Consider eliminating **E** in the example…

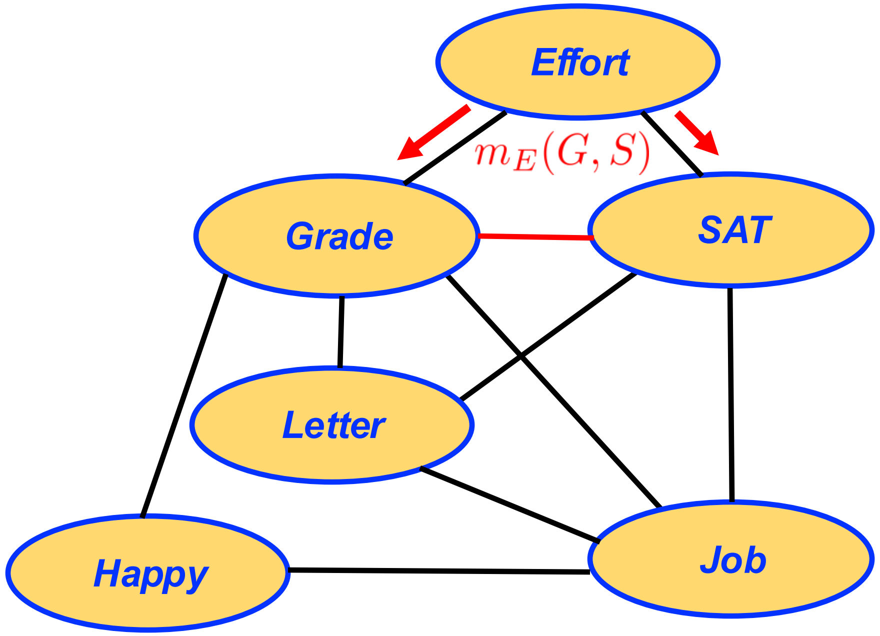
$$m_E(G,S) = \sum_e \boxed{m_D(G,e)\psi(e)\psi(S,e)}$$

Multiplication creates intermediate factor:

$$\phi(S,G,E) = m_D(G,E)\psi(E)\psi(S,E)$$

Assuming all variables are K-valued, new factor $\phi(S,G,E)$ has K^3 entries requiring

$$\mathcal{O}(K^3)$$



$$P(E,H,G,S,L,J) \propto m_D(G,E)\psi(E)$$
$$\psi(S,E)\psi(L,G)\psi(J,S,L)\psi(H,J,G)$$

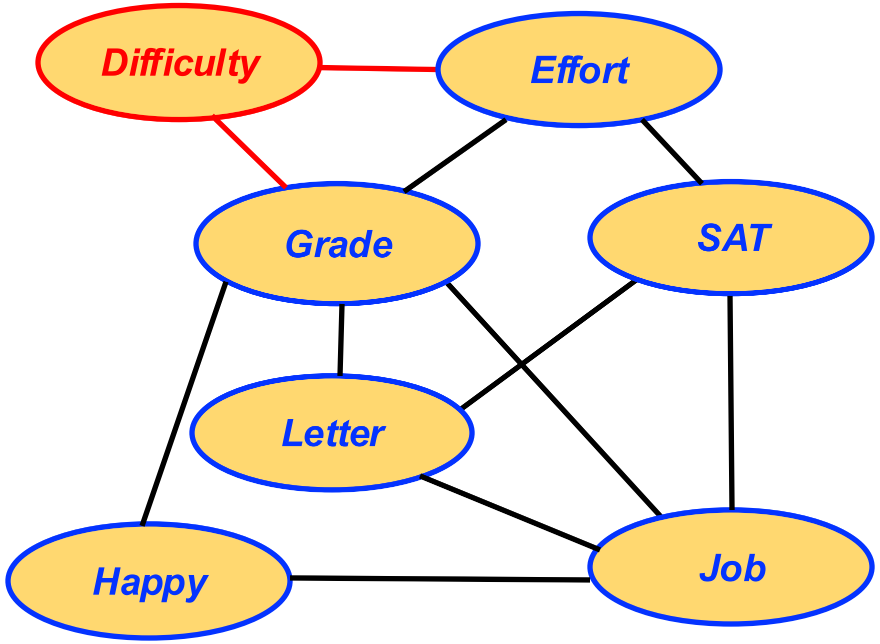
Complexity determined by size of the largest intermediate factor

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$\mathcal{O}(K^3)$



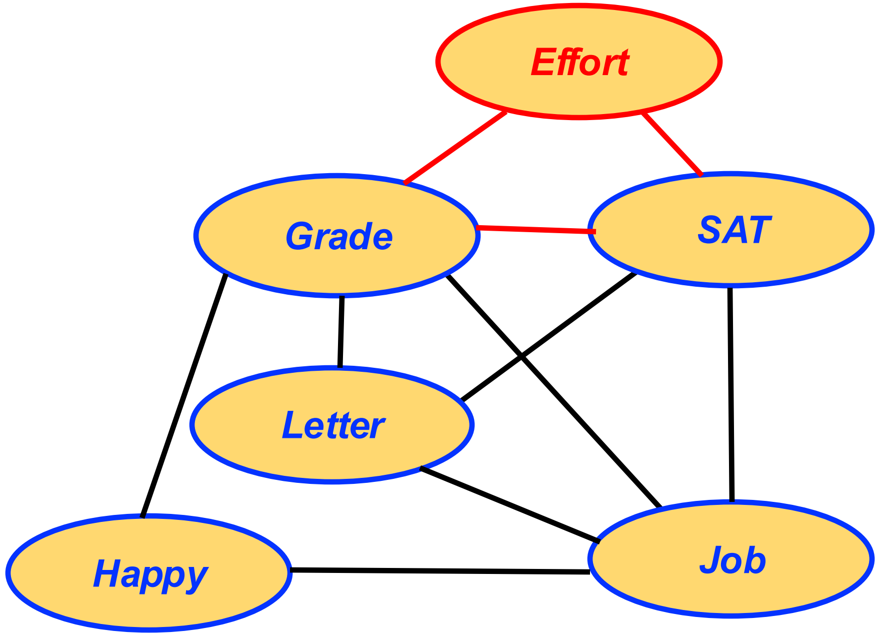
$$\phi(D, E, G) = \mathcal{O}(K^3)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$\mathcal{O}(K^3)$



$$\phi(E, G, S) = \mathcal{O}(K^3)$$

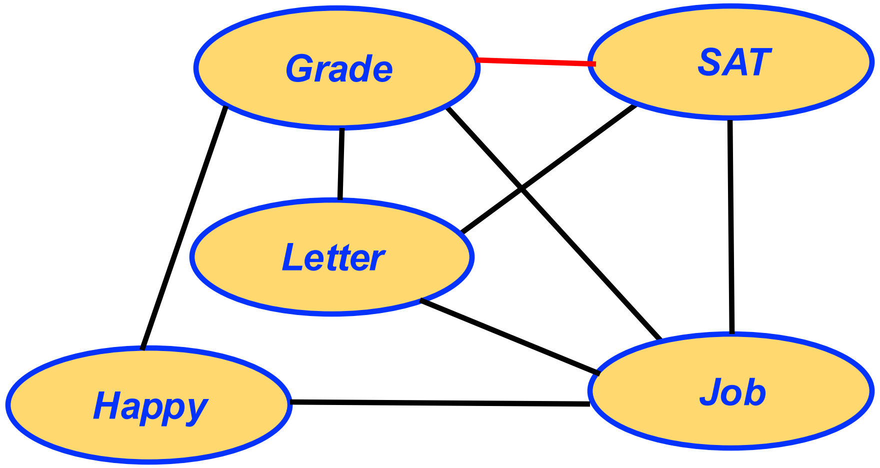
Computational Complexity

Elimination order D, E, H, G, S, L

Fill-in Edge

Worst-case Complexity:

$\mathcal{O}(K^3)$



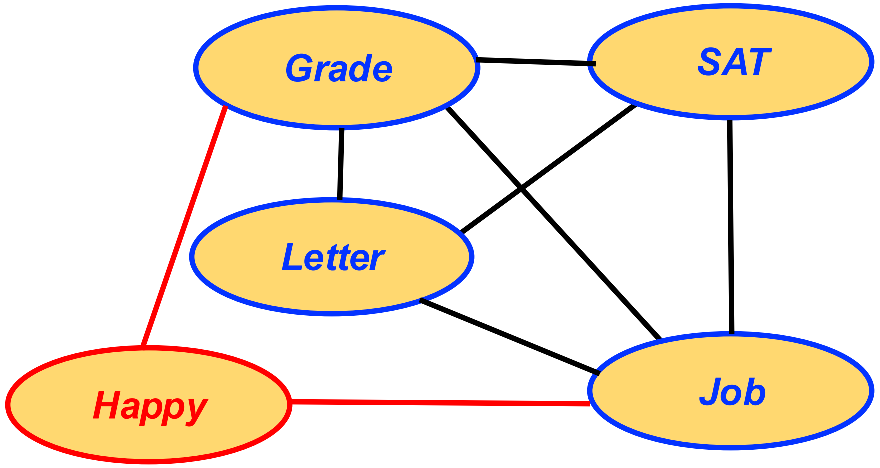
$$\phi(E, G, S) = \mathcal{O}(K^3)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$$\mathcal{O}(K^3)$$



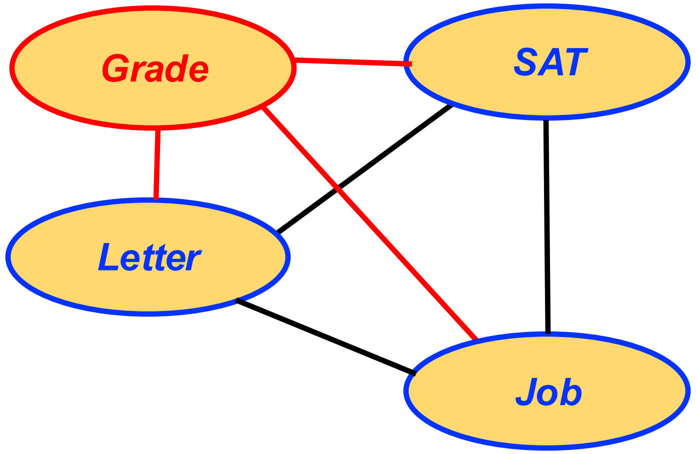
$$\phi(H, G, J) = \mathcal{O}(K^3)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$\mathcal{O}(K^4)$



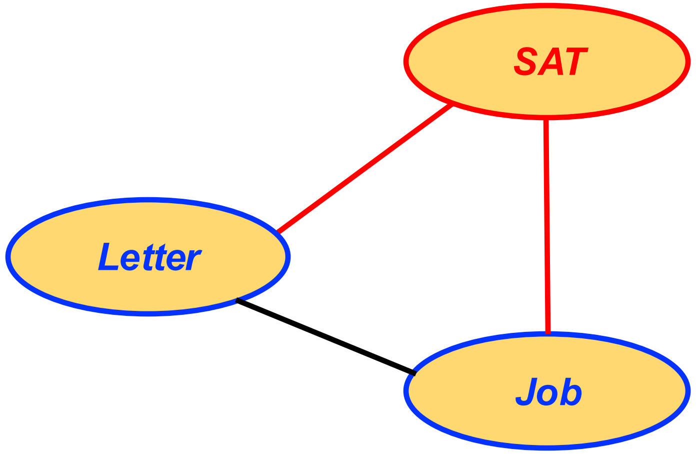
$$\phi(G, S, L, J) = \mathcal{O}(K^4)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$\mathcal{O}(K^4)$



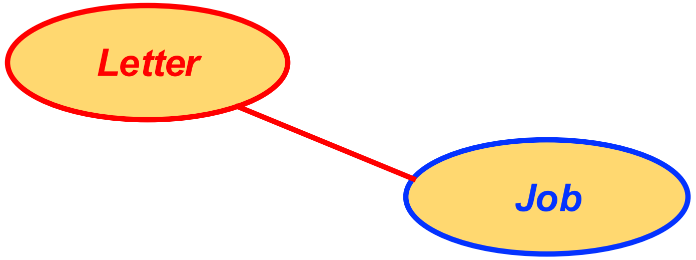
$$\phi(S, L, J) = \mathcal{O}(K^3)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$\mathcal{O}(K^4)$



$$\phi(L, J) = \mathcal{O}(K^2)$$

Computational Complexity

Elimination order D, E, H, G, S, L

Worst-case Complexity:

$$\mathcal{O}(K^4)$$

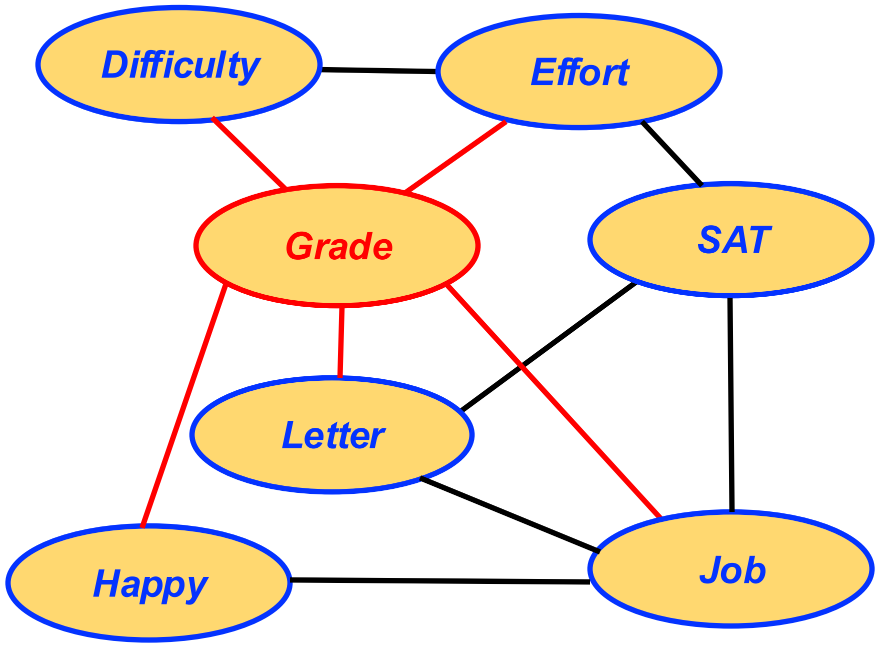
What if we choose a different elimination order?

Job

$$\phi(L, J) = \mathcal{O}(K^2)$$

Computational Complexity

Eliminate G first…



$$\phi(G, D, E, L, H, J) = \mathcal{O}(K^6)$$

Worst-case Complexity:

$\mathcal{O}(K^6)$

Complexity depends on elimination order…

For N variables worst case is:

$\mathcal{O}(K^N)$

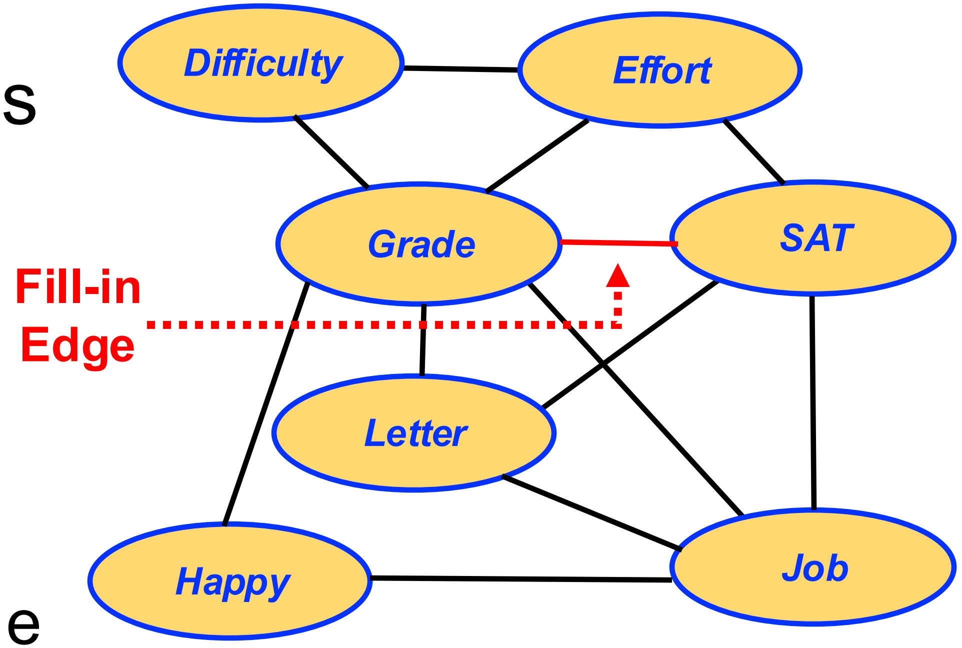
Optimal Ordering

The *induced graph* is the union of all graphs generated running variable elimination:

e.g. ordering D, E, H, G, S, L

Theorem (Informally) Given some elimination ordering:

1. Scope of every factor generated during variable elimination **is a clique in the induced graph**
2. Every **maximal clique** in the induced graph is a scope of some intermediate factor (of var. elim.)



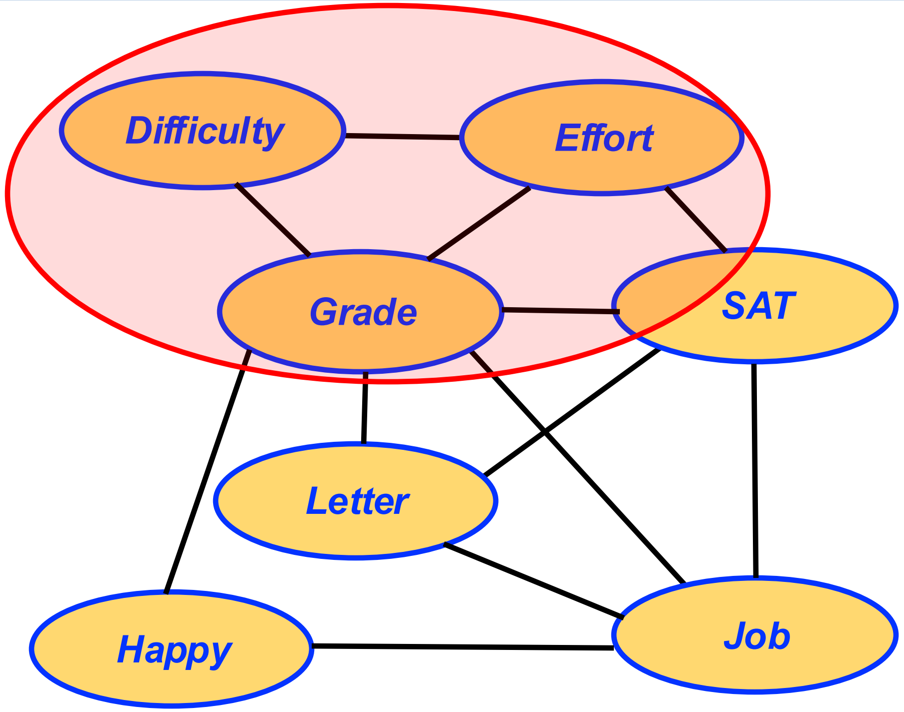
Induced graph cliques ⟷ **Intermediate factors**

Induced graph (and complexity) depend strongly on elimination order

Optimal Ordering

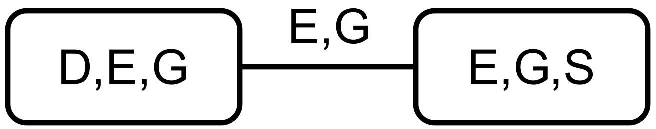
Clique Tree

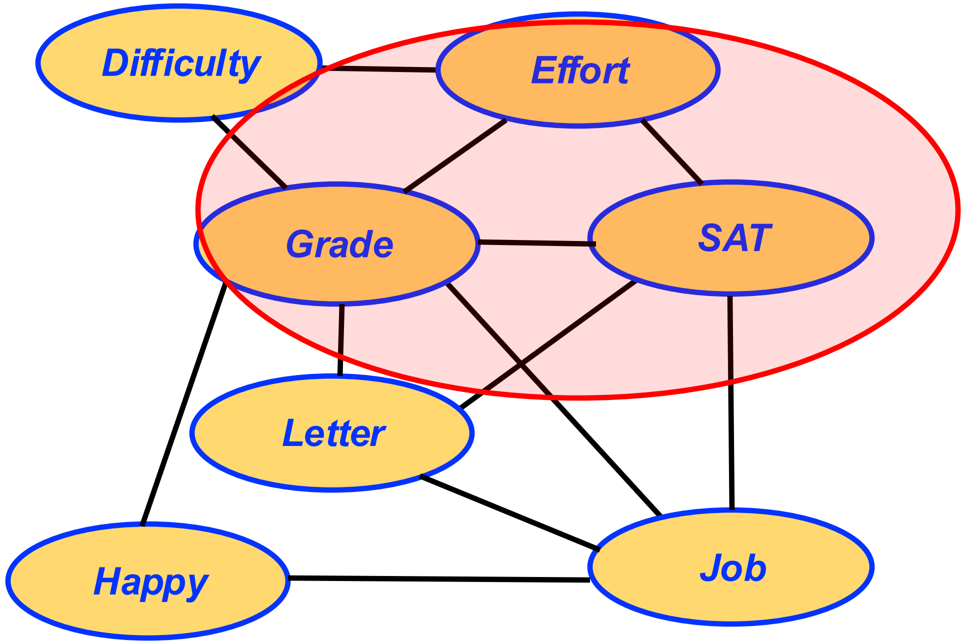
D,E,G



Optimal Ordering

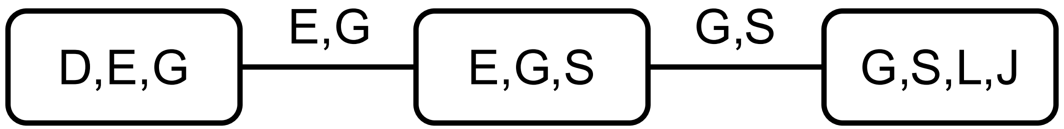
Clique Tree





Optimal Ordering

Clique Tree



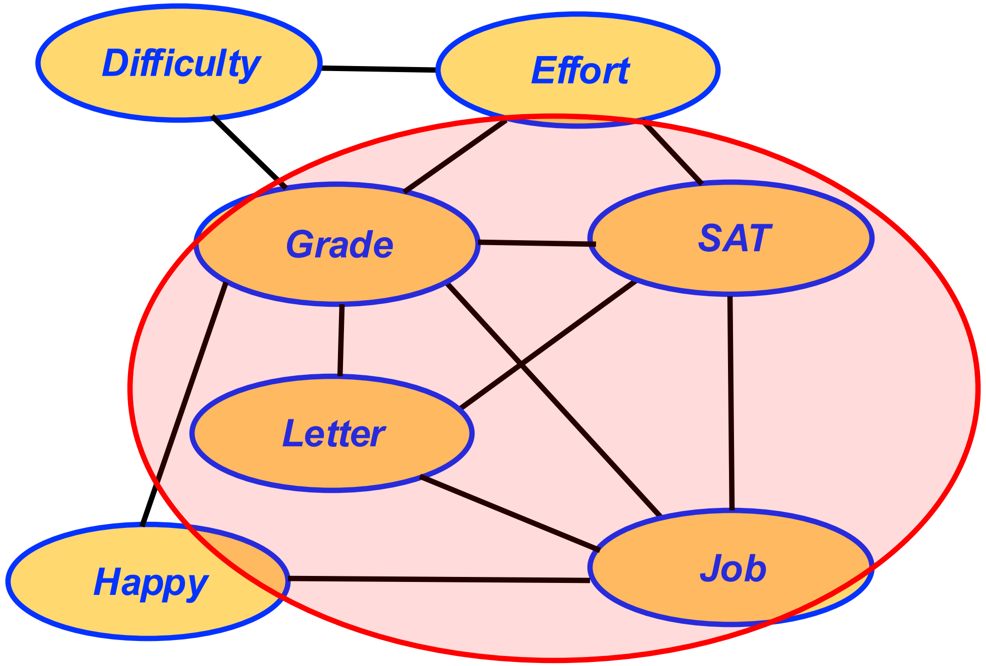
D,E,G

E,G

E,G,S

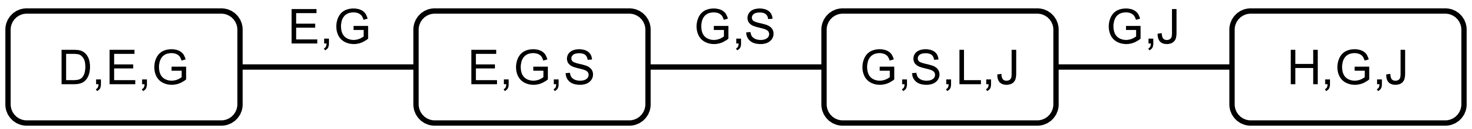
G,S

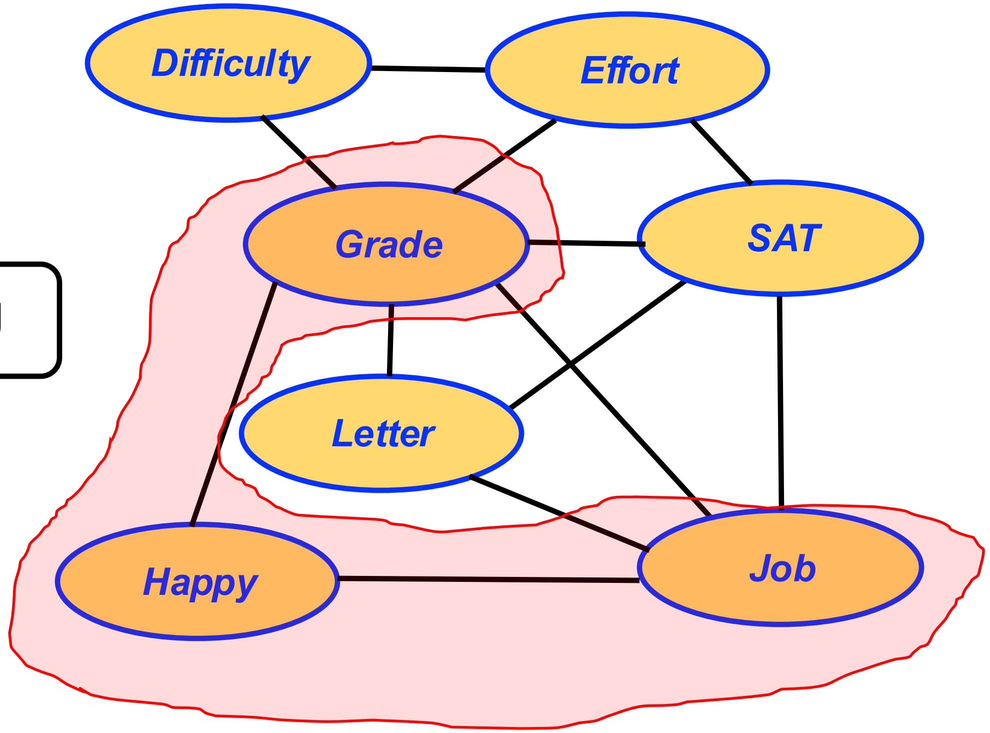
G,S,L,J



Optimal Ordering

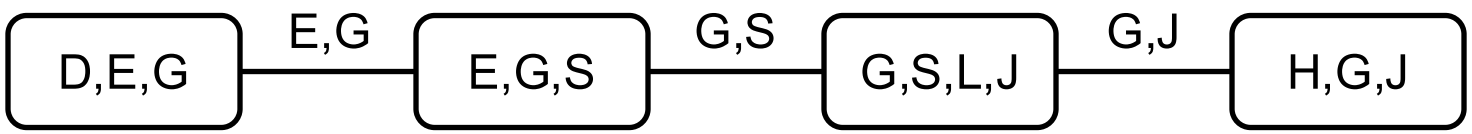
Clique Tree





Optimal Ordering

Clique Tree



D,E,G

E,G

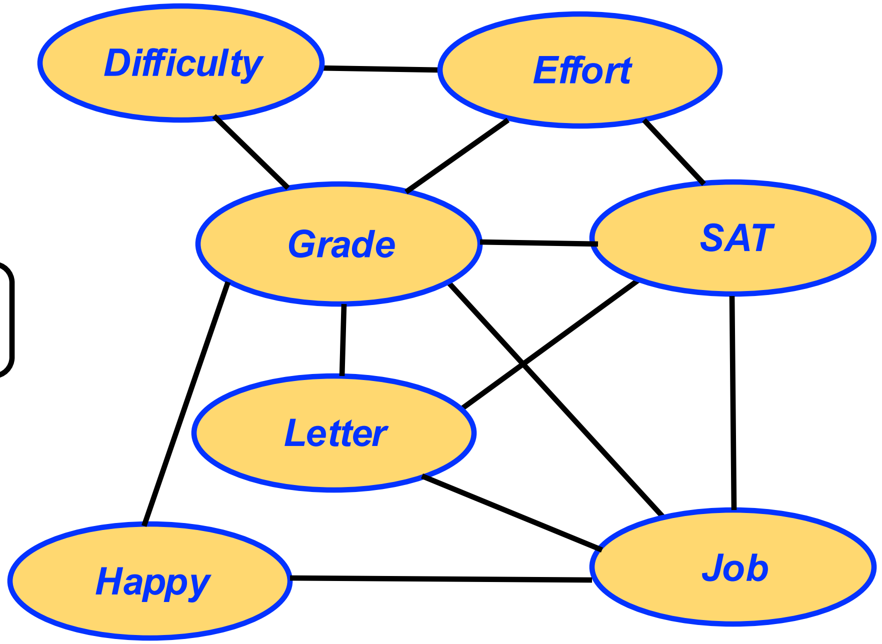
E,G,S

G,S

G,S,L,J

G,J

H,G,J



Elimination order $\prec$ induces graph with maximal cliques $\mathcal{C}(\prec)$ and *width:*

$$w(\prec) = \max_{c \in \mathcal{C}(\prec)} |c| - 1$$

- Complexity of variable elimination is $\mathcal{O}(K^{w(\prec)+1})$
- Lowest complexity given by the *treewidth:*

$$w^* = \min_{\prec} \max_{c \in \mathcal{C}(\prec)} |c| - 1$$

It is NP-hard to compute treewidth, and therefore an optimal elimination order (of course…)

Variable Elimination Summary

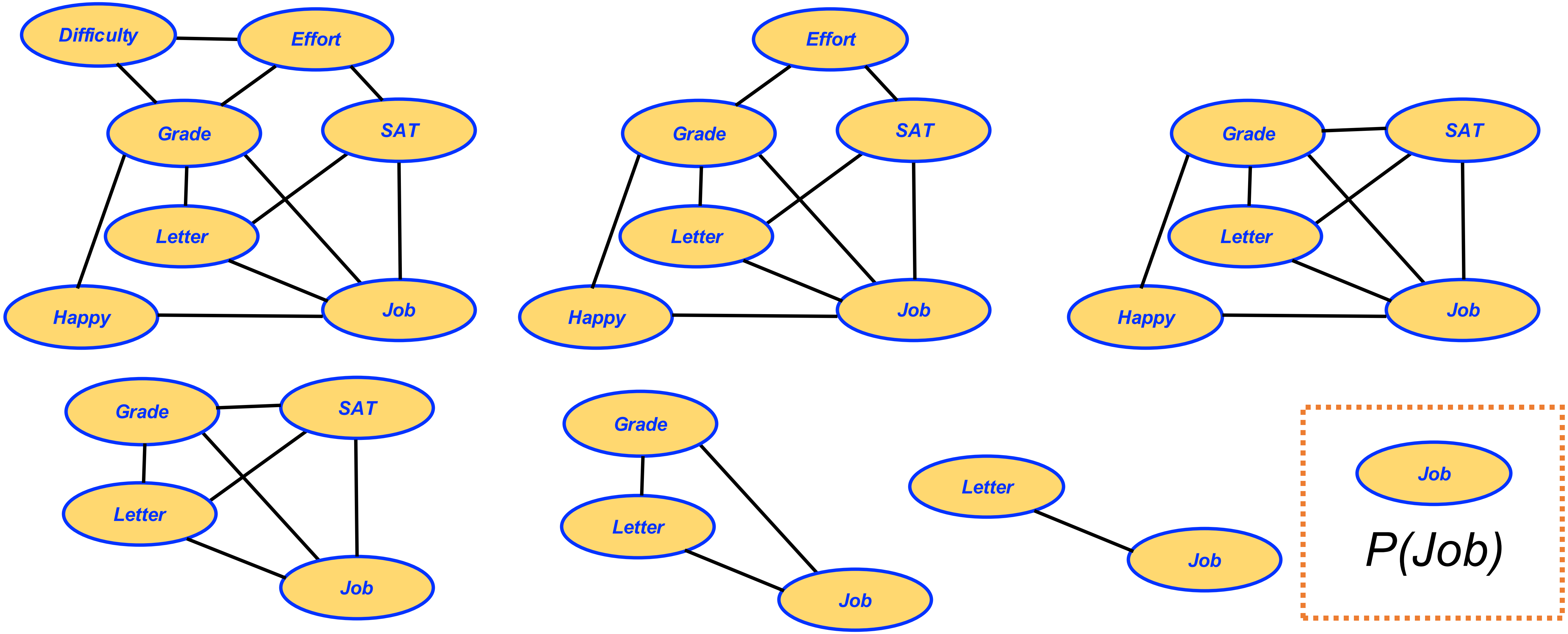
- Variable elimination allows computation of marginals / conditionals
- Algorithm is valid for **any graphical model**
- Suffices to show variable elimination for MRFs, since Bayes nets → MRFs by *moralization*
- Worst-case complexity is dependent on elimination order, and is **exponential** in number of variables
- Optimal ordering = treewidth, is NP-hard to compute

Outline

- ➢ Sum-Product Belief Propagation
- ➢ Loopy Belief Propagation
- ➢ Variable Elimination
- ➢ Junction Tree Algorithm
- ➢ Max-Product Belief Propagation

Variable Elimination

Recall variable elimination sequentially marginalizes out variables…



Variable Elimination

Two major limitations of variable elimination:

1. Computation **exponential** in size of the largest intermediate factor (equivalently, largest clique in clique tree)

2. Computation is not reused for computing a series of marginals

E.g. Suppose we use variable elimination to compute a marginal on an **HMM** with T nodes, each being K-valued

- It takes $\mathcal{O}(TK^2)$ time to compute a single marginal
- It takes $\mathcal{O}(T^2K^2)$ time to compute **all marginals**
- We know forward-backward computes all marginals in $\mathcal{O}(TK^2)$

Marginal Inference Algorithms

	One Marginal	***All Marginals***
Tree	Elimination applied to leaves of tree	Belief Propagation (BP) or sum-product algorithm
Graph	Variable Elimination	Junction Tree Algorithm BP on a junction tree (special clique tree)

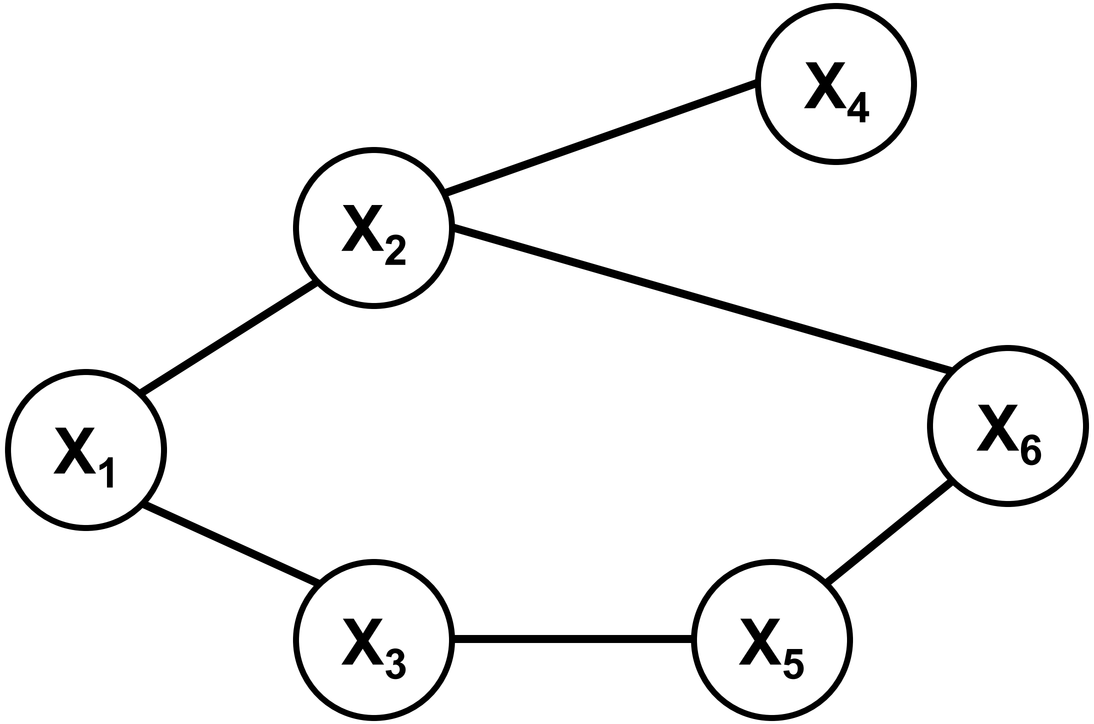
Marginal Inference Algorithms

	One Marginal	***All Marginals***
Tree	Elimination applied to leaves of tree	Belief Propagation (BP) or sum-product algorithm
Graph	Variable Elimination	Junction Tree Algorithm BP on a junction tree (special clique tree)

Clique Tree

Elimination order: 6,5,4,3,2,1

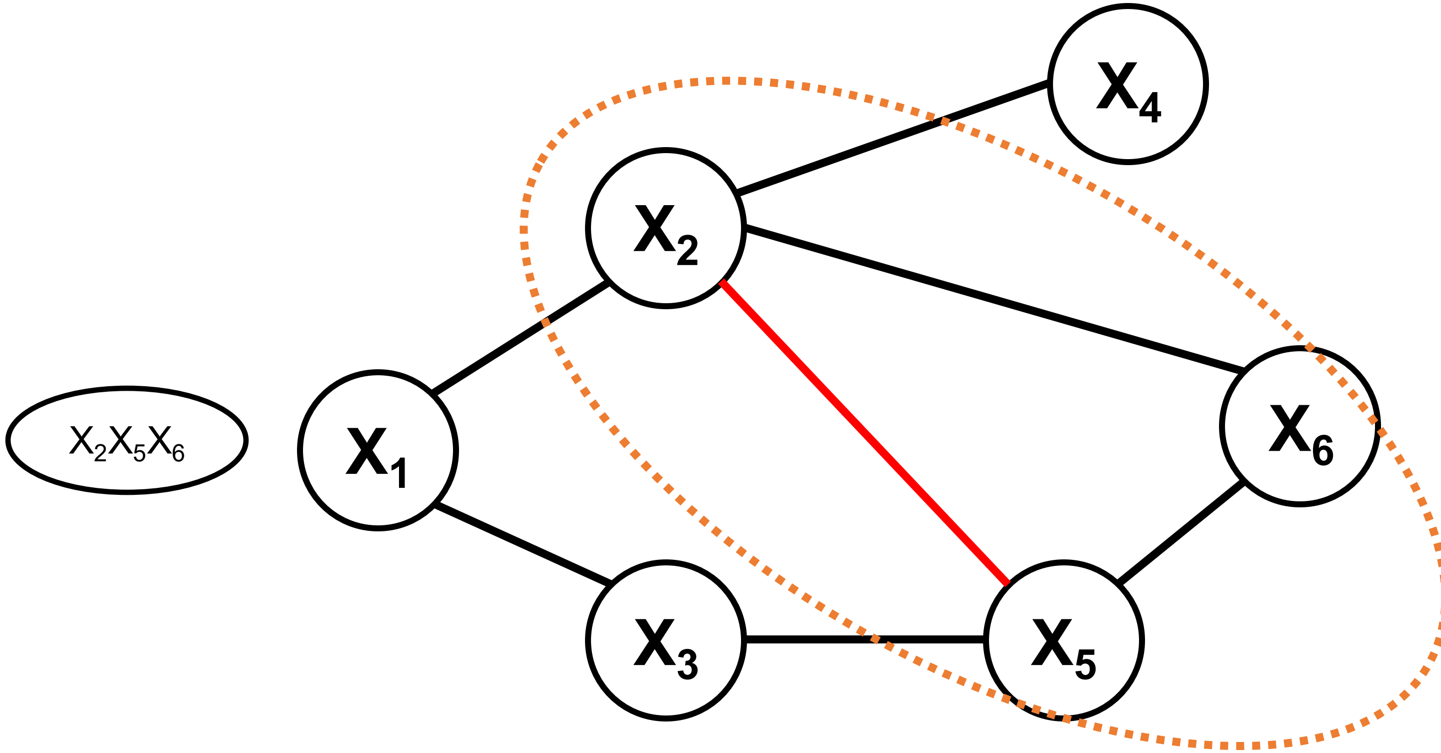
Clique Tree



Clique Tree

Elimination order: 6,5,4,3,2,1

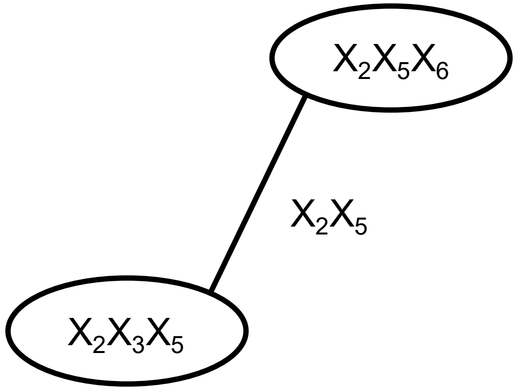
Clique Tree

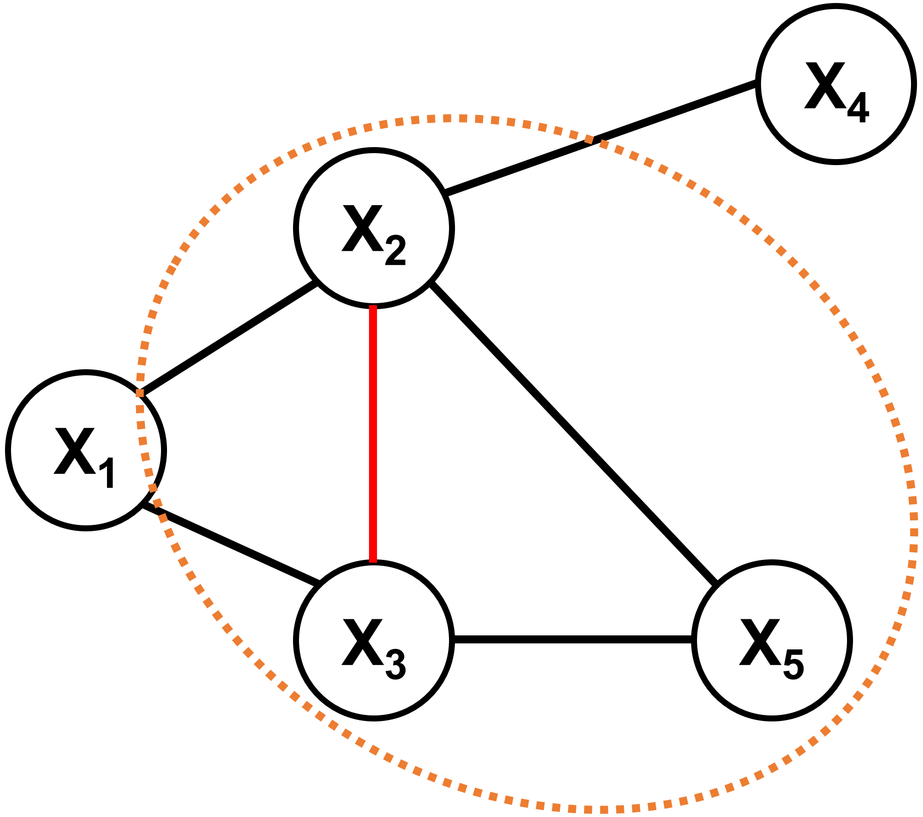


Clique Tree

Elimination order: 6,5,4,3,2,1

Clique Tree

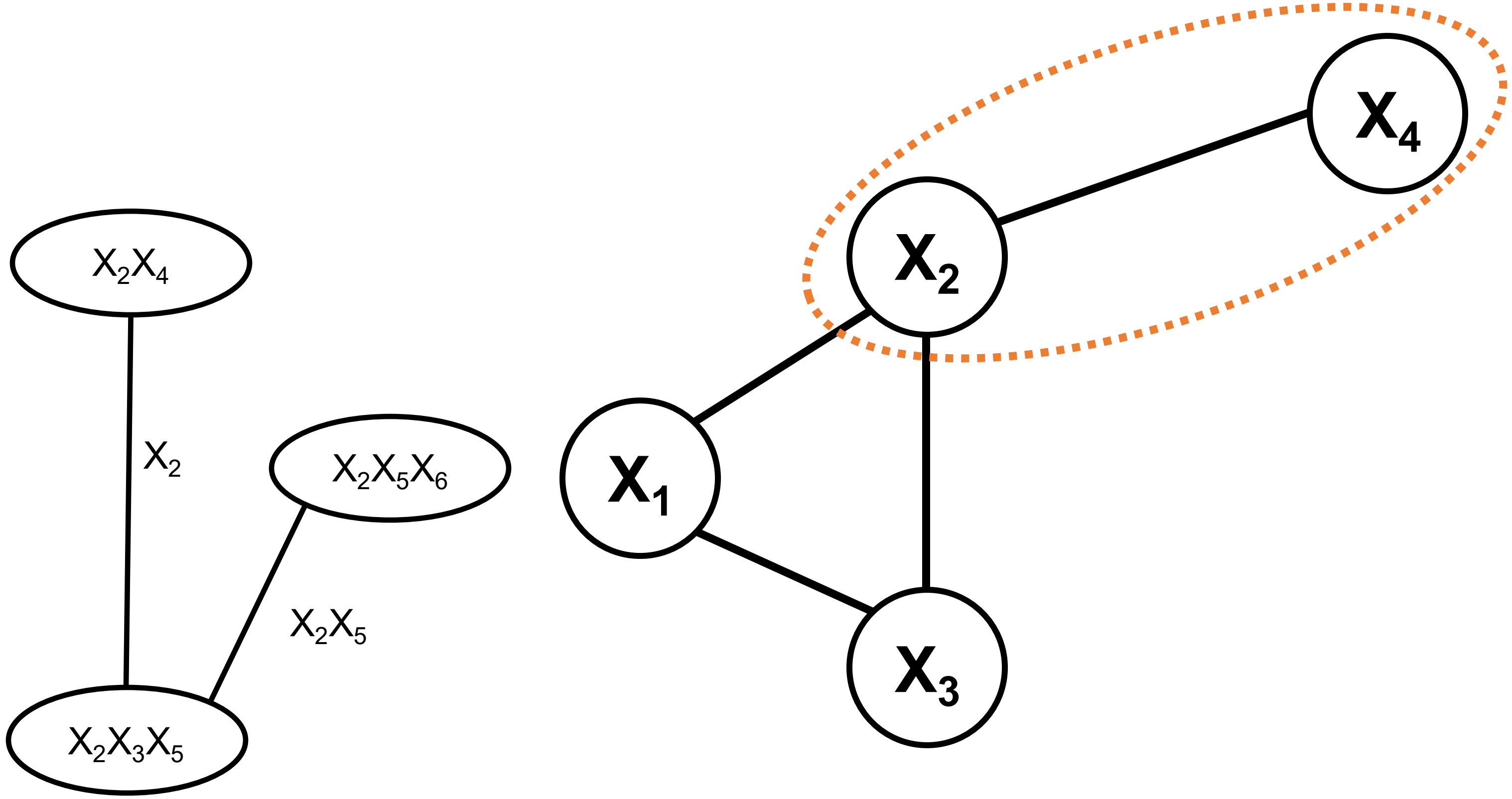




Clique Tree

Elimination order: 6,5,4,3,2,1

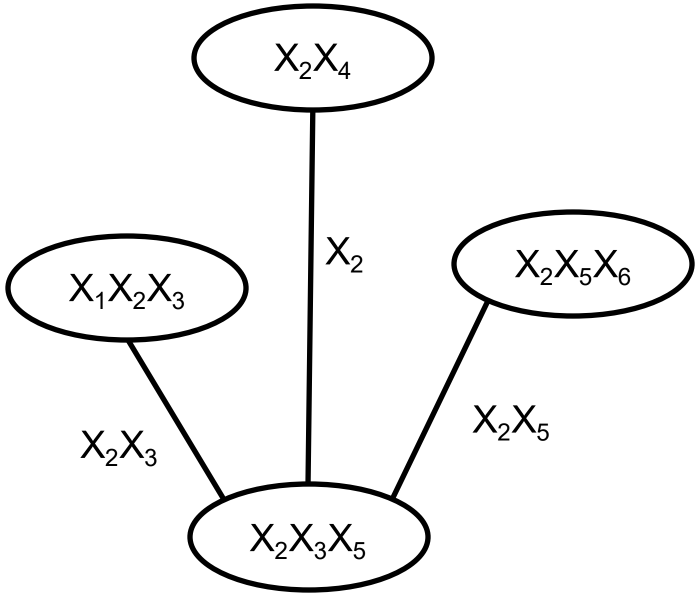
Clique Tree

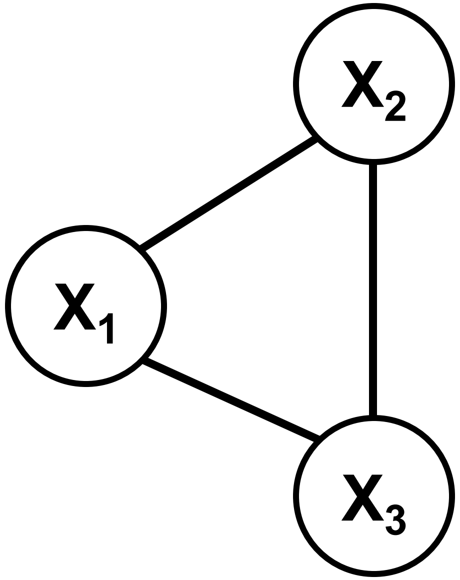


Clique Tree

Elimination order: 6,5,4,3,2,1

Clique Tree

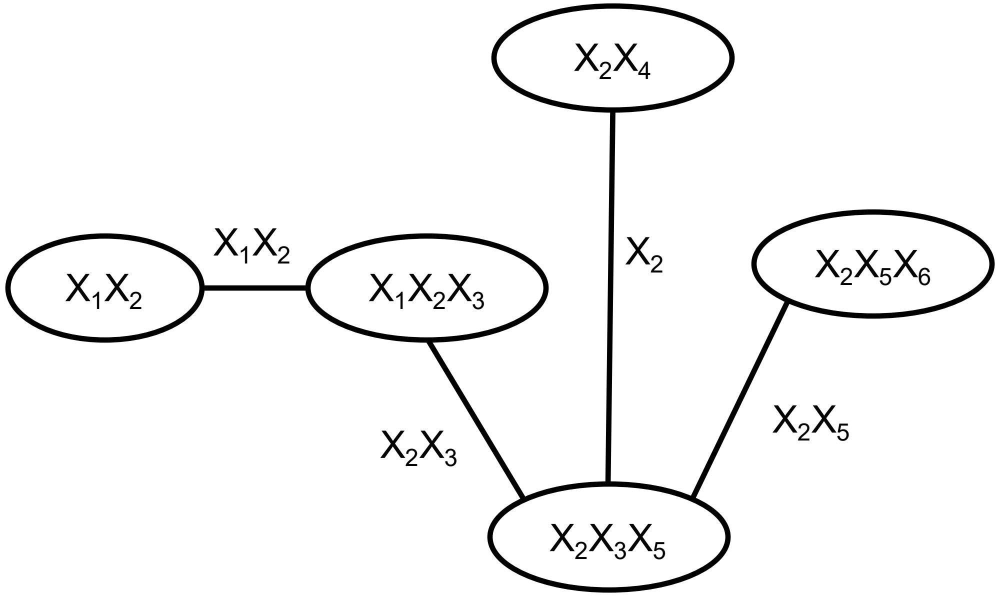


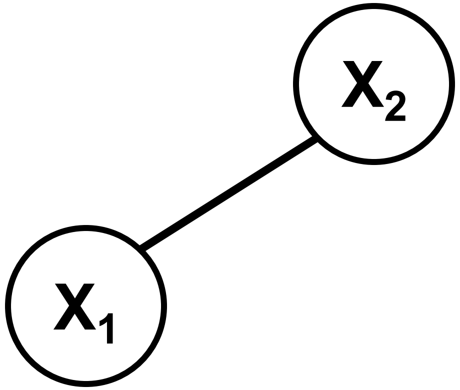


Clique Tree

Elimination order: 6,5,4,3,2,1

Clique Tree

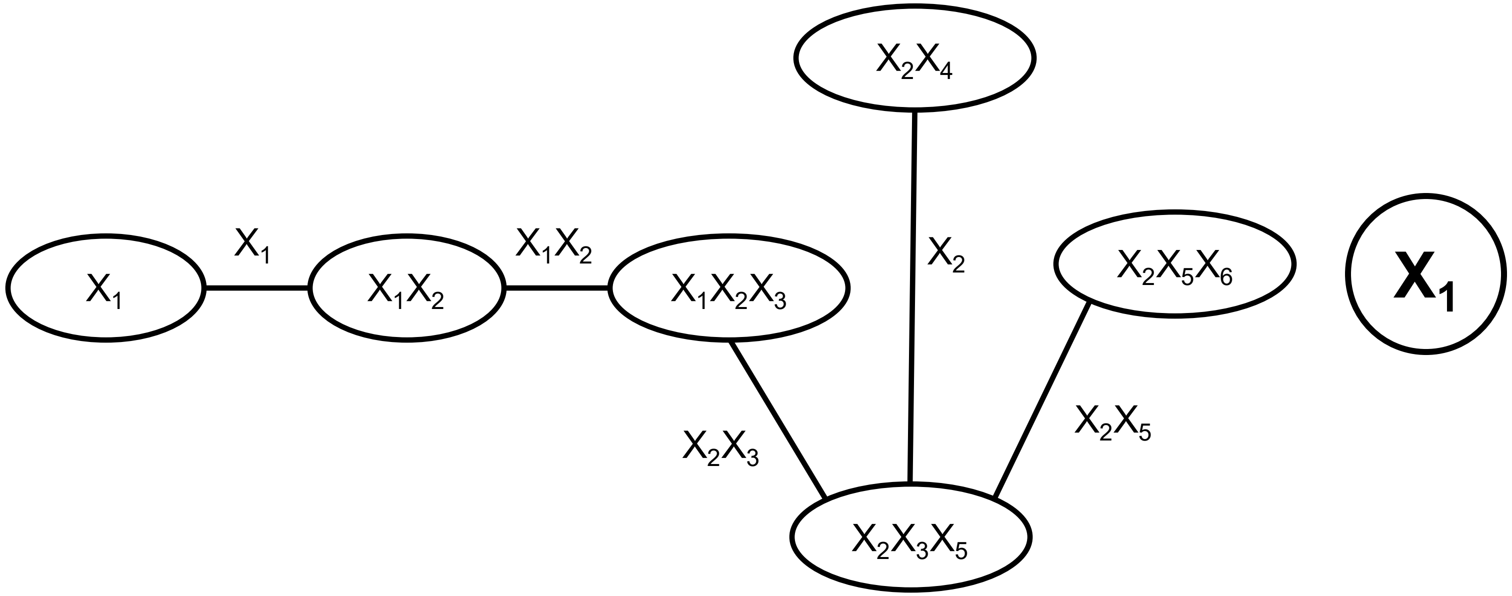




Clique Tree

Elimination order: 6,5,4,3,2,1

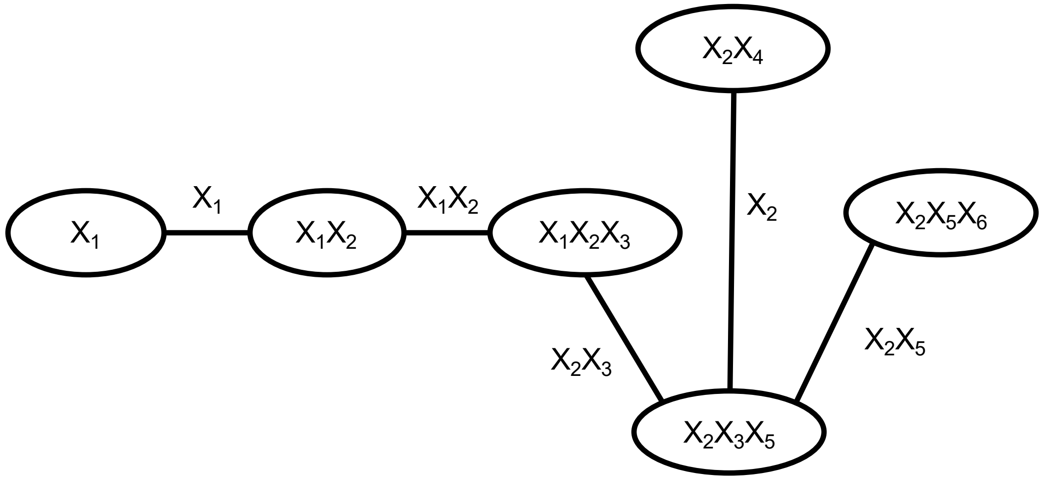
Clique Tree



Junction Tree

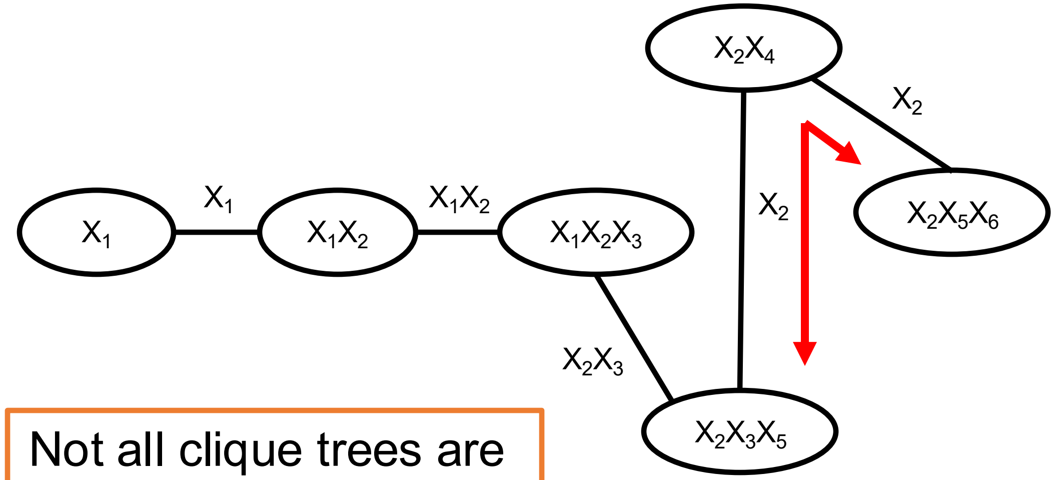
Definition (Running intersection) For any pair of clique nodes V,W all cliques on the *unique path* between V and W contain shared variables

Junction Tree



Not A Junction Tree

$$\{X_2, X_3, X_5\} \cap \{X_2, X_5, X_6\} = \{X_2, X_5\}$$



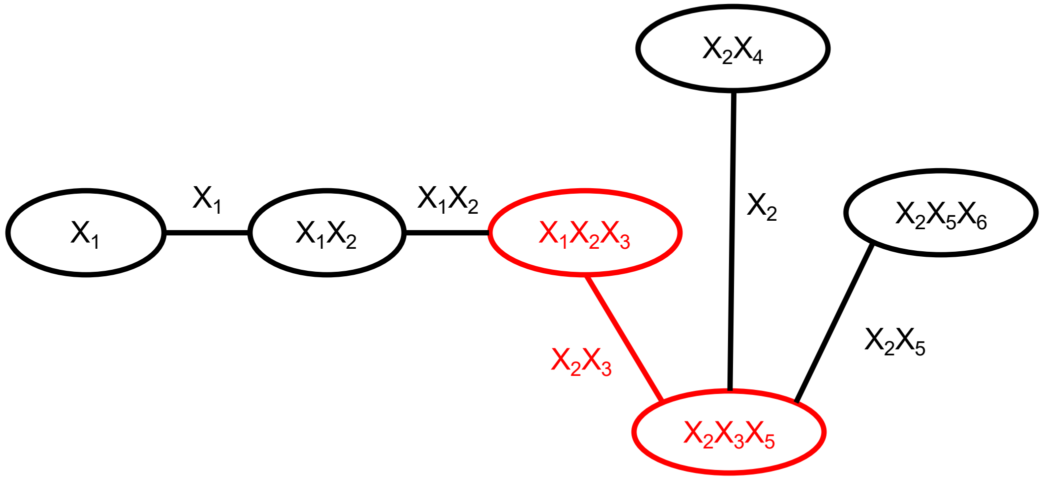
Not all clique trees are junction trees

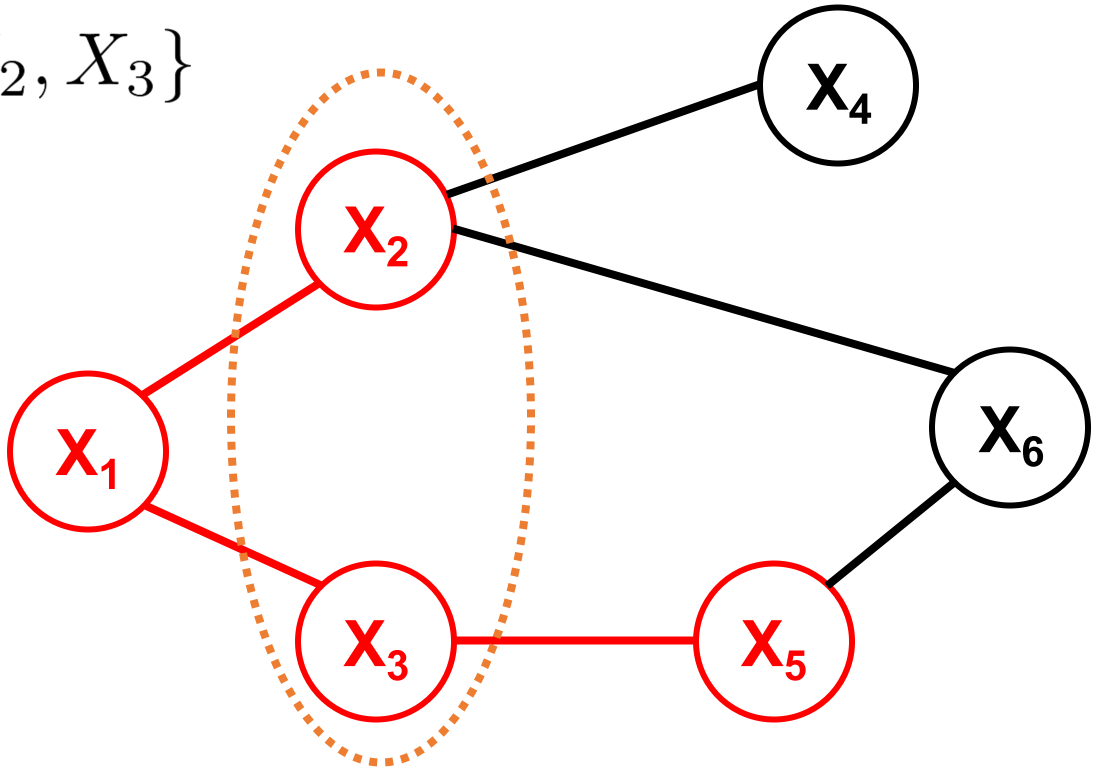
A **junction tree** is a clique tree with the **running intersection** property

Junction Tree

Clique tree edges are separator sets in original MRF…so clique tree encodes conditional independencies

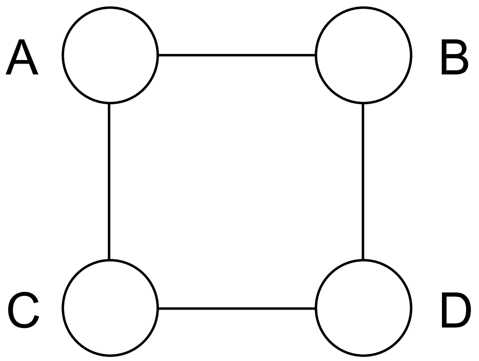
$$X_1 \perp X_5 \mid \{X_2, X_3\}$$

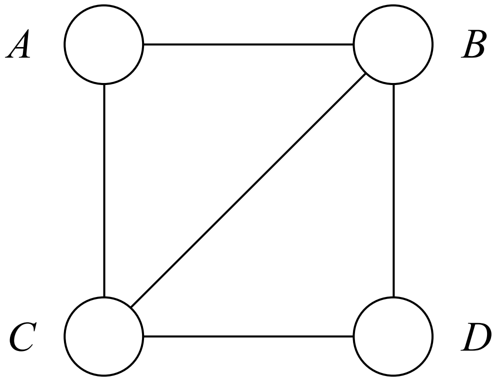


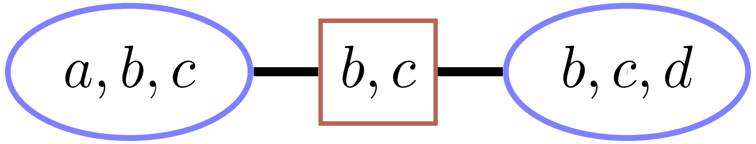


Theorem A clique tree resulting from variable elimination **satisfies the running intersection property** and is thus **a junction tree**

Junction Trees and Triangulation







- A *chord* is an edge connecting two non-adjacent nodes in some *cycle*
- A cycle is *chordless* if it contains no chords
- A graph is *triangulated (chordal)* if it contains no chordless cycles of length 4 or more

Theorem: The maximal cliques of a graph have a corresponding junction tree *if and only if* that undirected graph is triangulated

Lemma: *For a non-complete triangulated graph with at least 3 nodes, there is a decomposition of the nodes into disjoint sets A, B, S such that S separates A from B, and S is complete.*

- ➢ Key induction argument in constructing junction tree from triangulation
- ➢ Implies existence of *elimination ordering which introduces no new edges*

Induced Graph

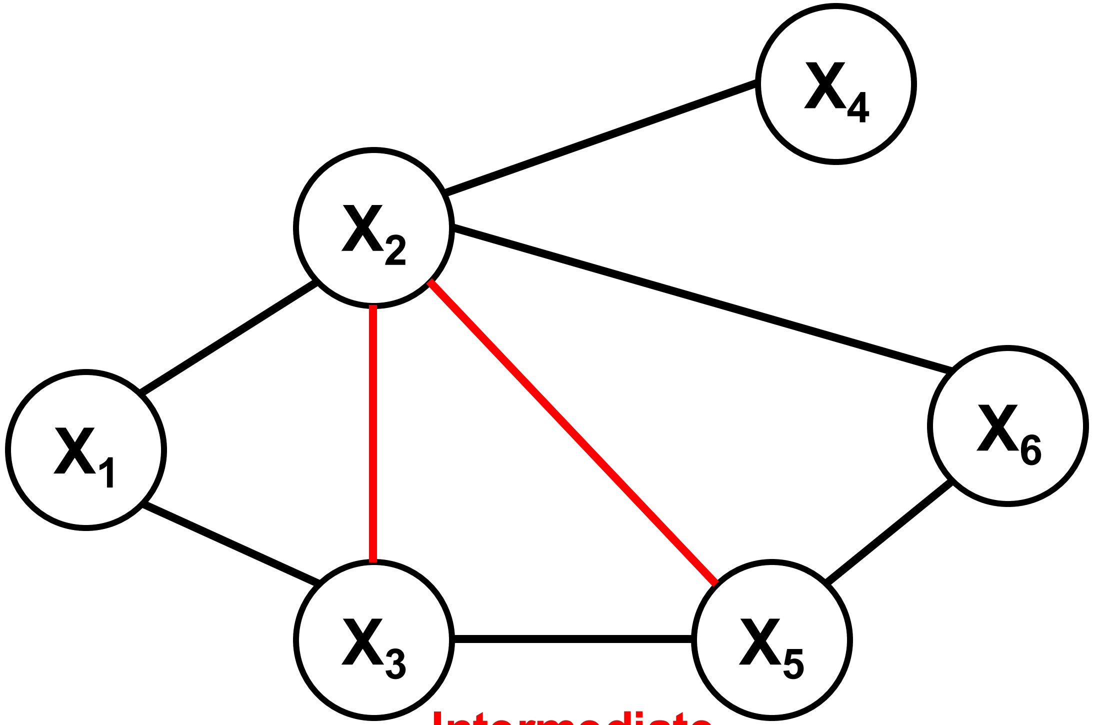
Recall the **induced graph** is the union over intermediate graphs from running variable elimination

The induced graph **is chordal** thus:

- Maximal cliques of the induced graph form a junction tree
- It admits an elimination ordering that introduces *no new edges*

Logic of junction tree algorithm:

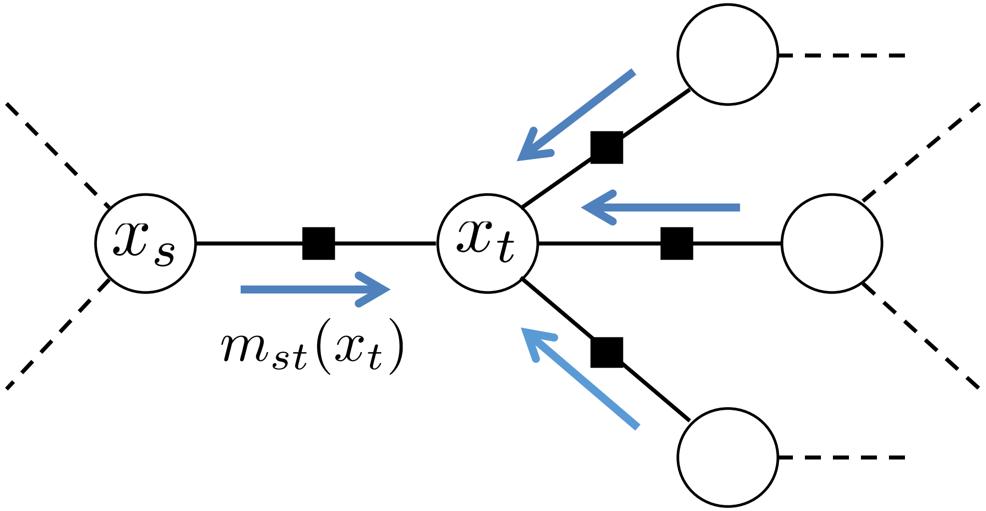
1. Triangulate the graph
   a. Implies a junction tree
   b. Induces an elimination order
2. Run sum-product BP on junction tree to compute **all clique marginals**

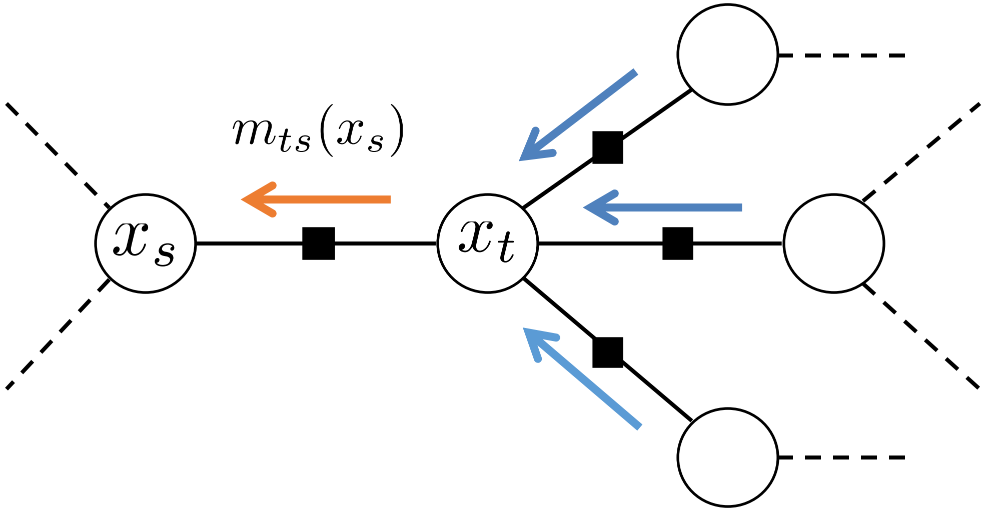


Intermediate Factor Edges

Reminder: Pairwise Sum-Product BP

Set of *neighbors* of node *t*: $\Gamma(t) = \{s \in \mathcal{V} \mid (s,t) \in \mathcal{F}\}$





$$p_t(x_t) \propto \prod_{s \in \Gamma(t)} m_{st}(x_t)$$

$$m_{ts}(x_s) = \sum_{x_t} \psi_{st}(x_s, x_t) \prod_{u \in \Gamma(t) \setminus s} m_{ut}(x_t)$$

K_t ⟶ *number of discrete states for random variable x$_t$*

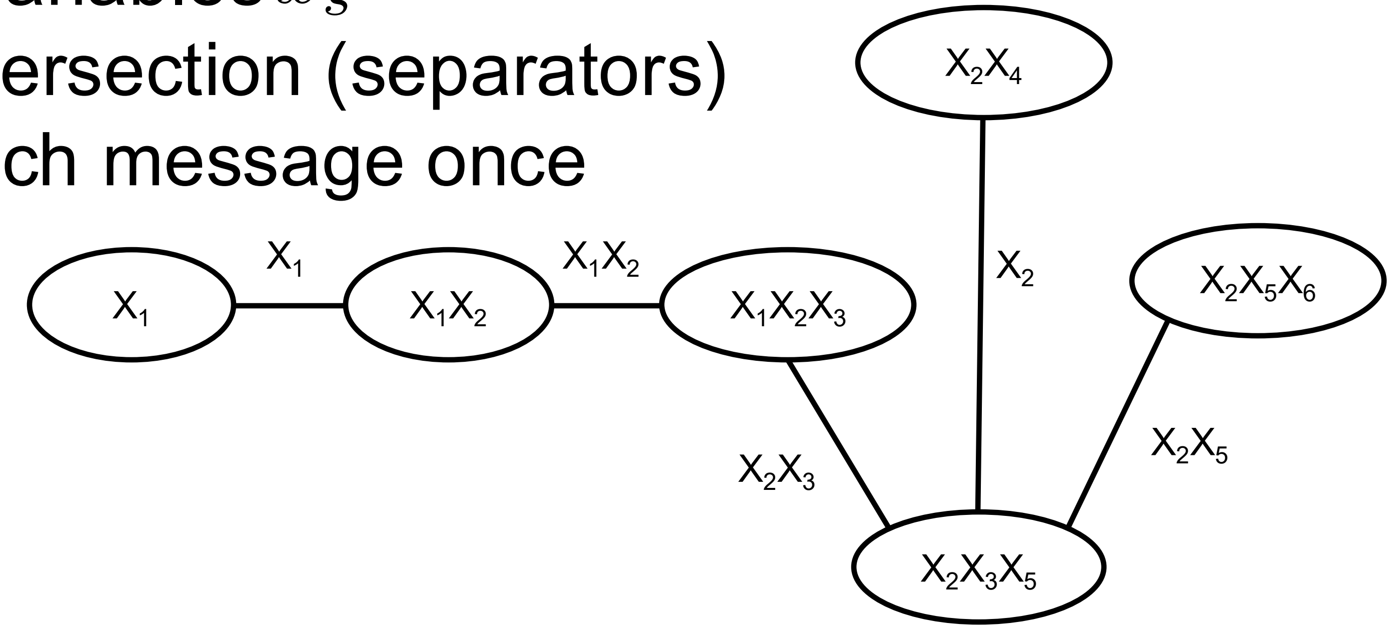
$p_t(x_t)$ ⟶ *marginal distribution of the K$_t$ discrete states of random variable x$_t$*

$m_{st}(x_t)$ ⟶ *message from node s to node t, a vector of K$_t$ non-negative numbers*

$m_{ts}(x_s)$ ⟶ *message from node t to node s, a vector of K$_s$ non-negative numbers*

Sum-Product for Junction Trees (Shafer-Shenoy)

- Express algorithm via original variables x_s
- Messages depend on clique intersection (separators)
- Efficient schedules compute each message once



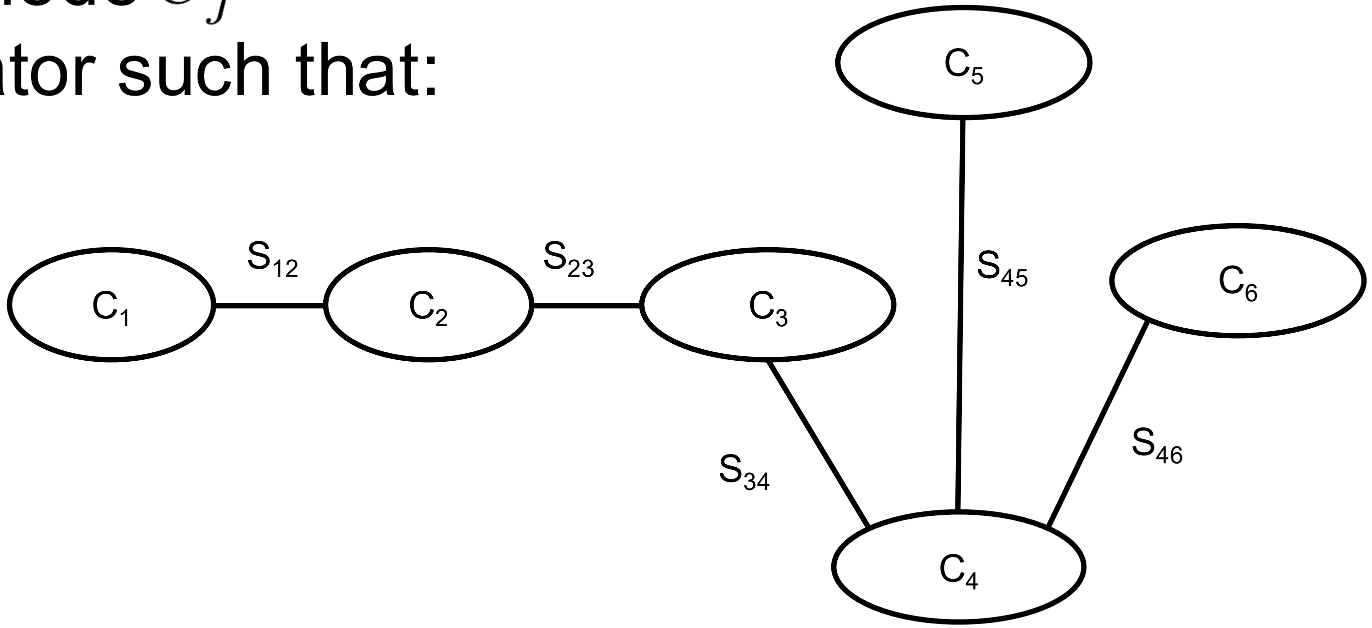
Sum-Product for Junction Trees (Shafer-Shenoy)

- Let x_{C_j} be variables in clique node C_j
- Let $x_{S_{ij}}$ be variables in separator such that:

$$x_{S_{ij}} = x_{C_i} \cap x_{C_j}$$

- Let *residual* variables be:

$$x_{R_{ij}} = x_{C_i} \backslash x_{S_{ij}}$$



Sum-Product for Junction Trees (Shafer-Shenoy)

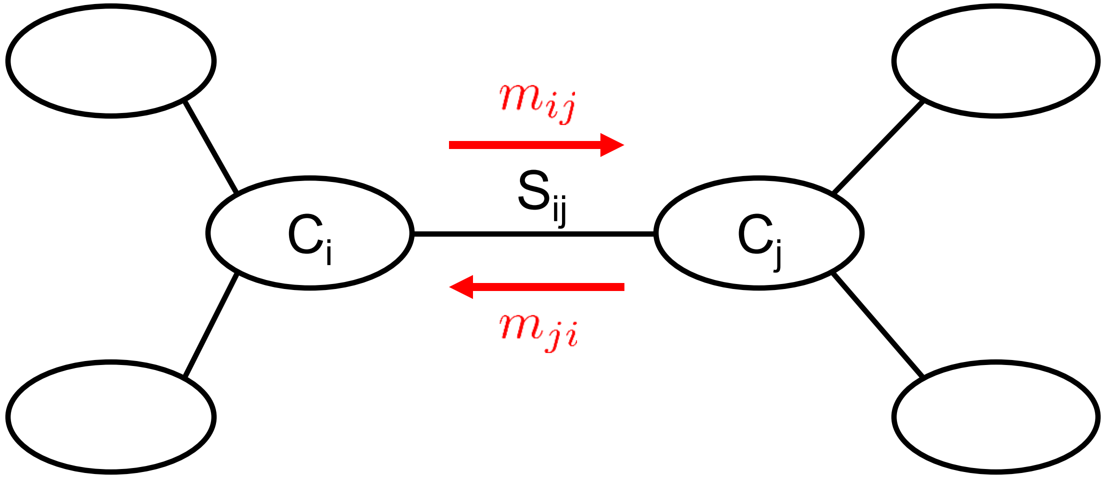
- Let x_{C_j} be variables in clique node C_j
- Let $x_{S_{ij}}$ be variables in separator such that:

$$x_{S_{ij}} = x_{C_i} \cap x_{C_j}$$

- Let *residual* variables be:

$$x_{R_{ij}} = x_{C_i} \backslash x_{S_{ij}}$$

- Pass sum-product messages between clique nodes

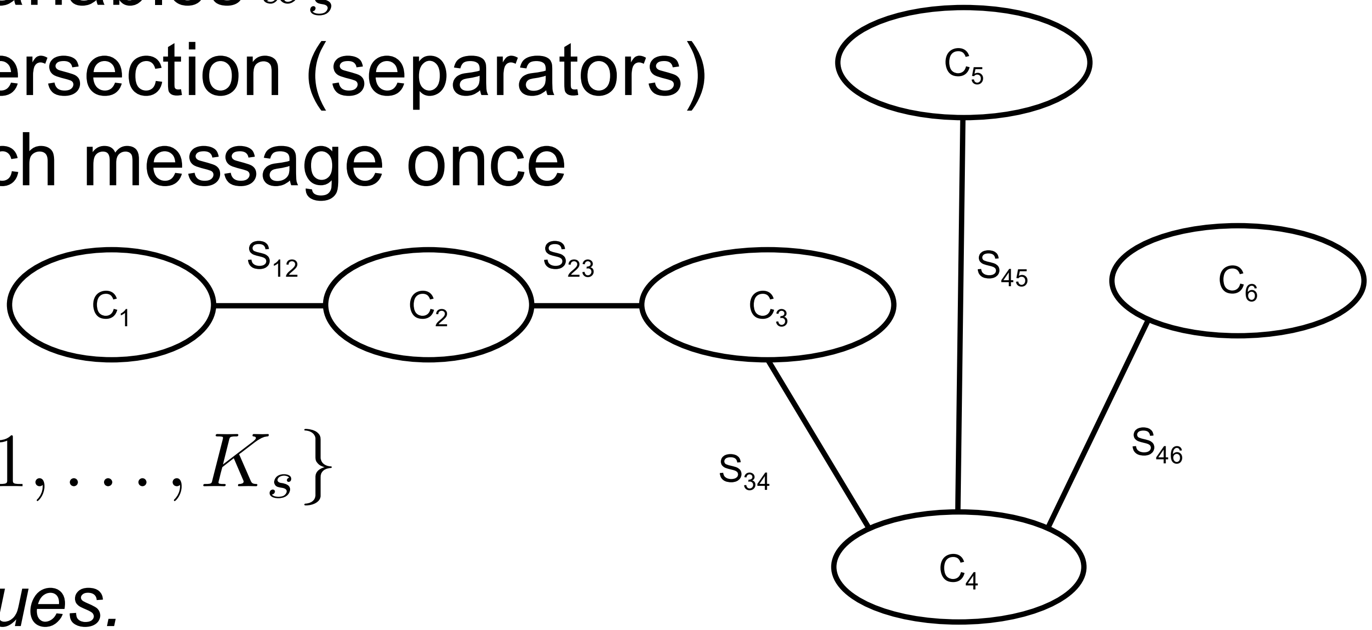


Message: $$m_{ji}(x_{S_{ji}}) \propto \sum_{x_{R_{ji}}} \psi_{C_j}(x_{C_j}) \prod_{k \in \Gamma(j) \backslash i} m_{kj}(x_{S_{kj}})$$

Marginal: $$p_j(x_{C_j}) \propto \psi_{C_j}(x_{C_j}) \prod_{i \in \Gamma(j)} m_{ij}(x_{S_{ij}})$$

Sum-Product for Junction Trees (Shafer-Shenoy)

- Express algorithm via original variables x_s
- Messages depend on clique intersection (separators)
- Efficient schedules compute each message once



Storage & Computational Cost

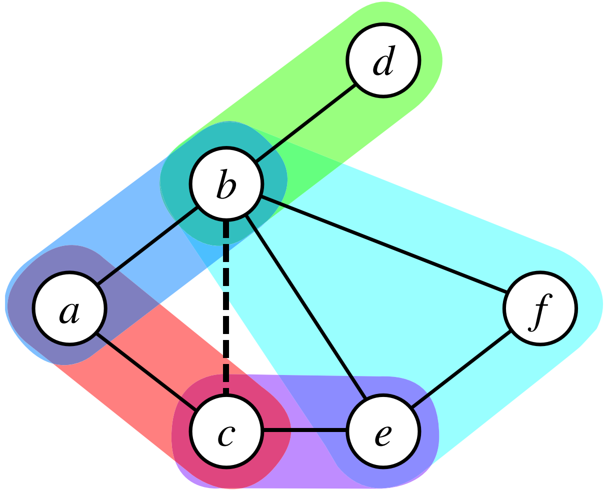
$$\mathcal{O}\left(\sum_j \prod_{s \in C_j} K_s\right), \text{ where } x_s \in \{1, \ldots, K_s\}$$

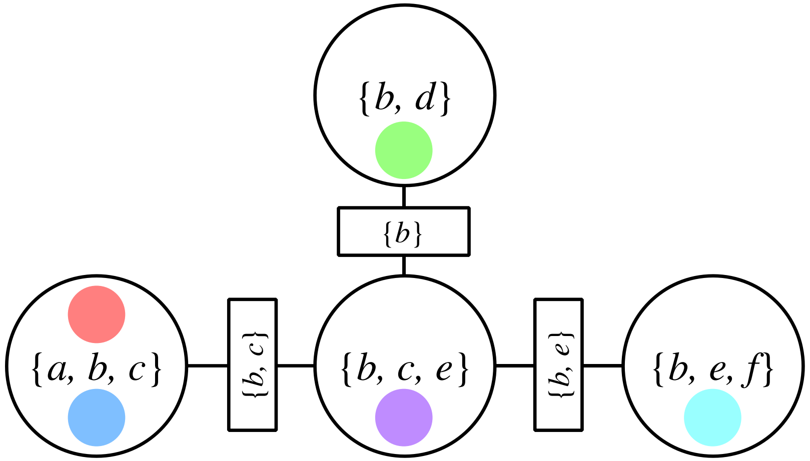
Exponential in sizes of maximal cliques.

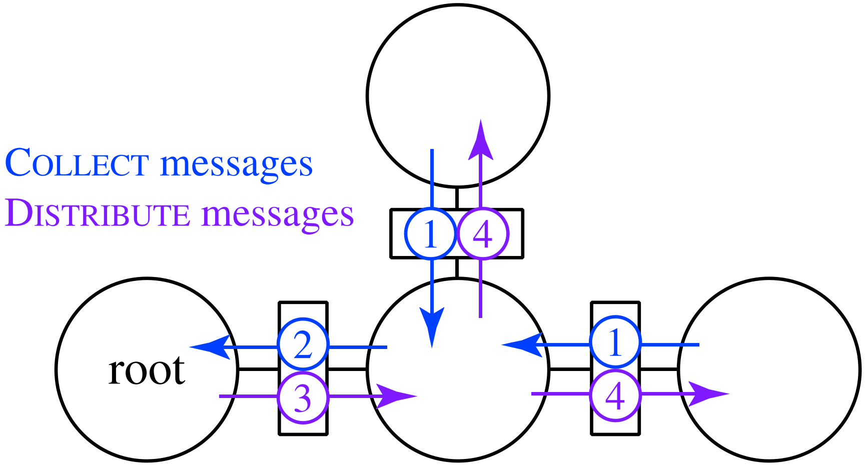
Message: $$m_{ji}(x_{S_{ji}}) \propto \sum_{x_{R_{ji}}} \psi_{C_j}(x_{C_j}) \prod_{k \in \Gamma(j) \setminus i} m_{kj}(x_{S_{kj}})$$

Marginal: $$p_j(x_{C_j}) \propto \psi_{C_j}(x_{C_j}) \prod_{i \in \Gamma(j)} m_{ij}(x_{S_{ij}})$$

Summary: Junction Tree Algorithm






COLLECT messages

DISTRIBUTE messages

$$p_j(x_{C_j}) \propto \psi_{C_j}(x_{C_j}) \prod_{i \in \Gamma(j)} m_{ij}(x_{S_{ij}})$$

$$S_{ij} = S_{ji} = C_i \cap C_j$$

Junction Tree Algorithms for General-Purpose Inference

1. If necessary, convert graphical model to undirected form (*linear in graph size*)
2. Triangulate the target undirected graph

➢ Any elimination ordering generates a valid triangulation (*linear in graph size*)

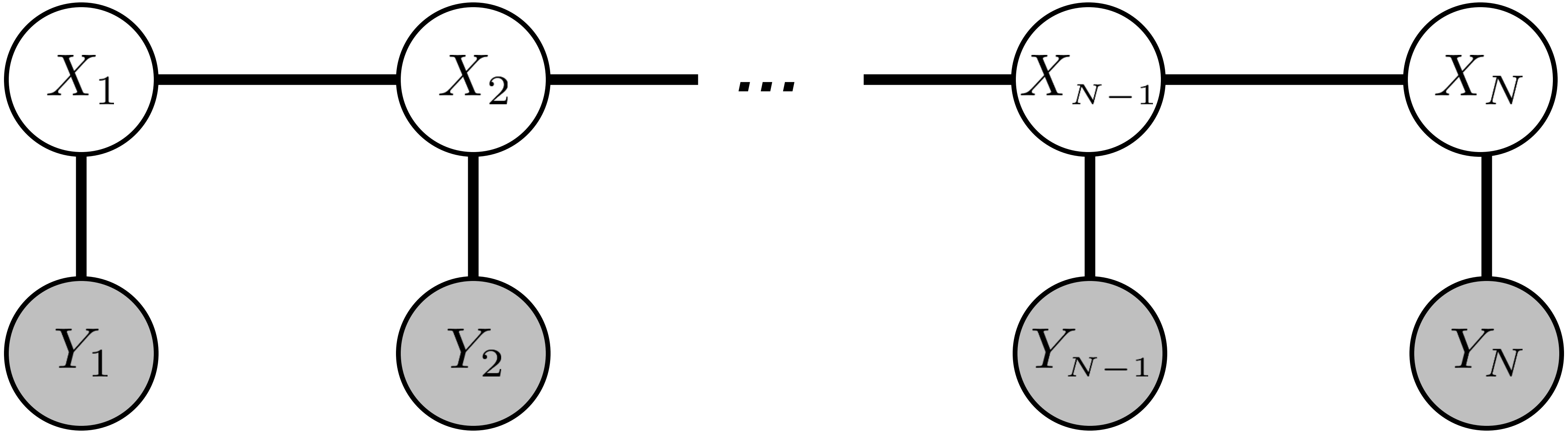
➢ Finding an optimal triangulation, with minimal cliques, is *NP-hard*

3. Arrange triangulated cliques into a junction tree (*at worst quadratic in graph size*)
4. Execute sum-product algorithm on junction tree (*exponential in clique size*)

Outline

- Sum-Product Belief Propagation
- Loopy Belief Propagation
- Variable Elimination
- Junction Tree Algorithm
- Max-Product Belief Propagation

Maximum A Posteriori (MAP) Inference



Rather than *marginalize* sometimes we want to *maximize, e.g.*

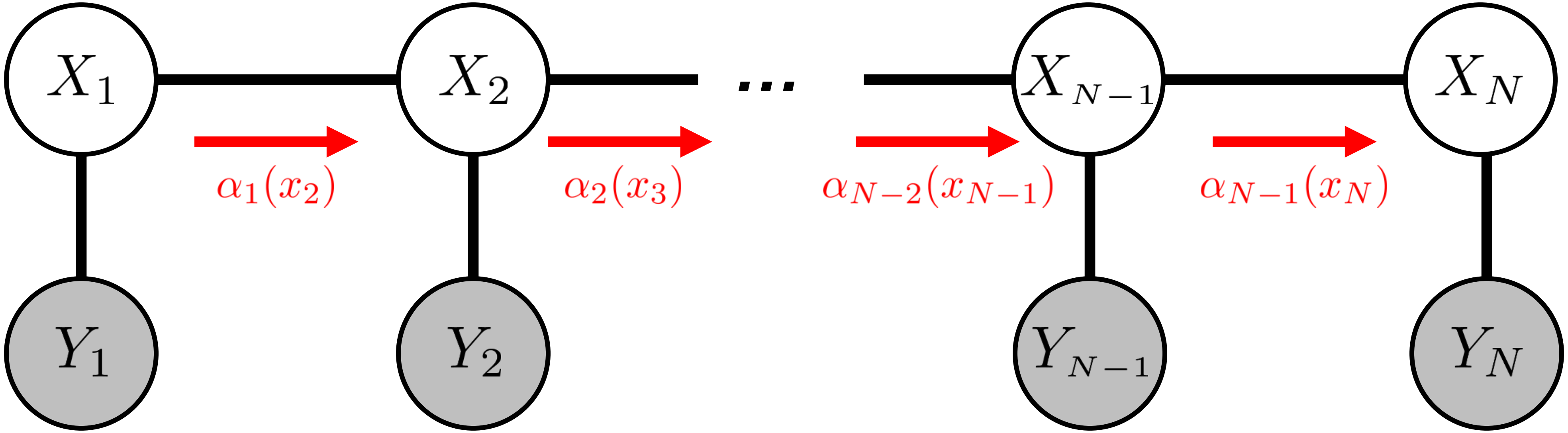
$$(x_1^*, x_2^*, \ldots, x_N^*) = \arg\max_{\mathbf{x}} p(\mathbf{x} \mid \mathbf{y})$$

Maximizing the log-joint is equivalent and numerically more stable:

$$(x_1^*, x_2^*, \ldots, x_N^*) = \arg\max_{\mathbf{x}} \log p(\mathbf{x}, \mathbf{y}) + \text{const.}$$

Forward-Backward Algorithm

Recall the Forward-Backward algorithm messages…

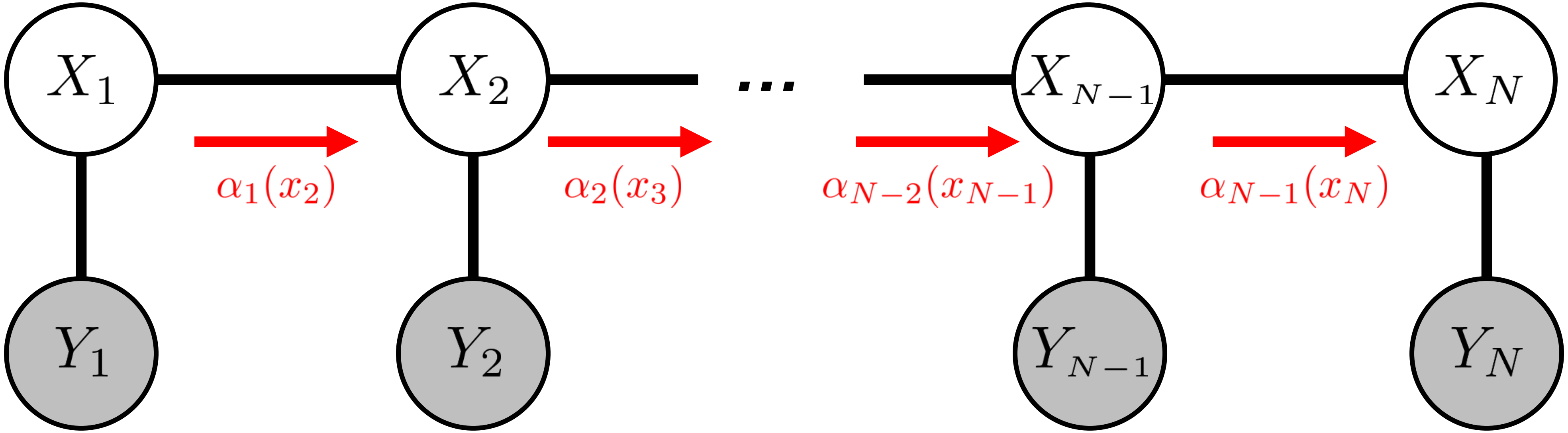


Forward message:

$$\alpha_{n-1}(x_n) = \underbrace{\textstyle\sum_{x_{n-1}} \alpha_{n-2}(x_{n-1})\psi(x_{n-1}, x_n)\psi(x_n, y_n)}_{\text{Sum over state } x_{n-1}}$$

Viterbi Algorithm

Maximize instead of marginalize…

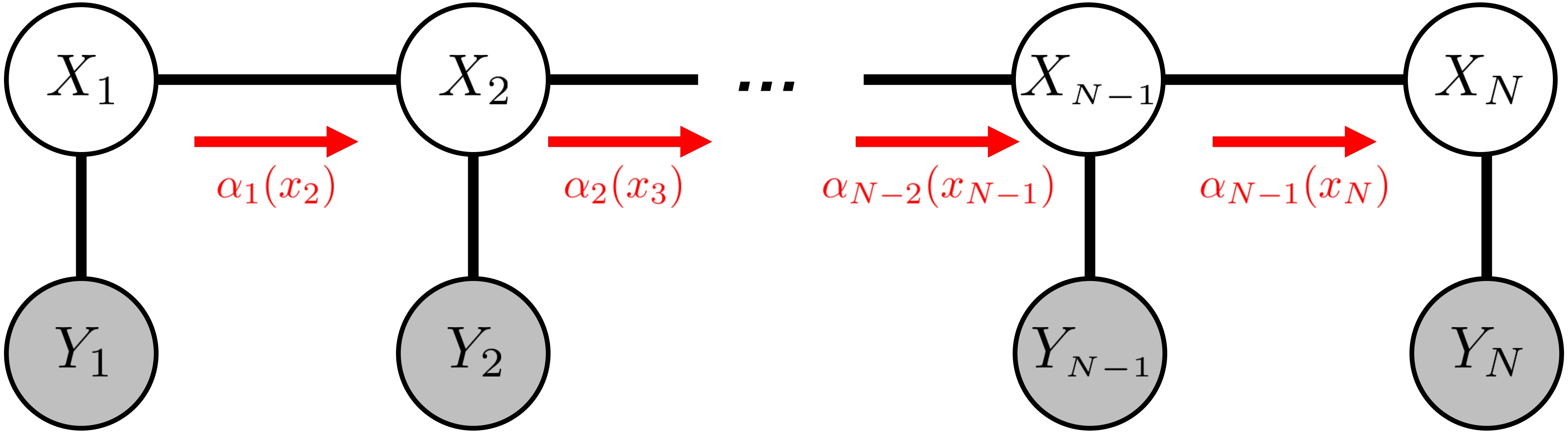


Forward message:

$$\alpha_{n-1}(x_n) = \underbrace{\max_{x_{n-1}} \log\psi(x_n, y_n) + \alpha_{n-2}(x_{n-1}) + \log\psi(x_{n-1}, x_n)}_{\text{Maximize over state } x_{n-1} \text{ (in log-domain)}}$$

Viterbi Algorithm

Maximize instead of marginalize…



Forward message:

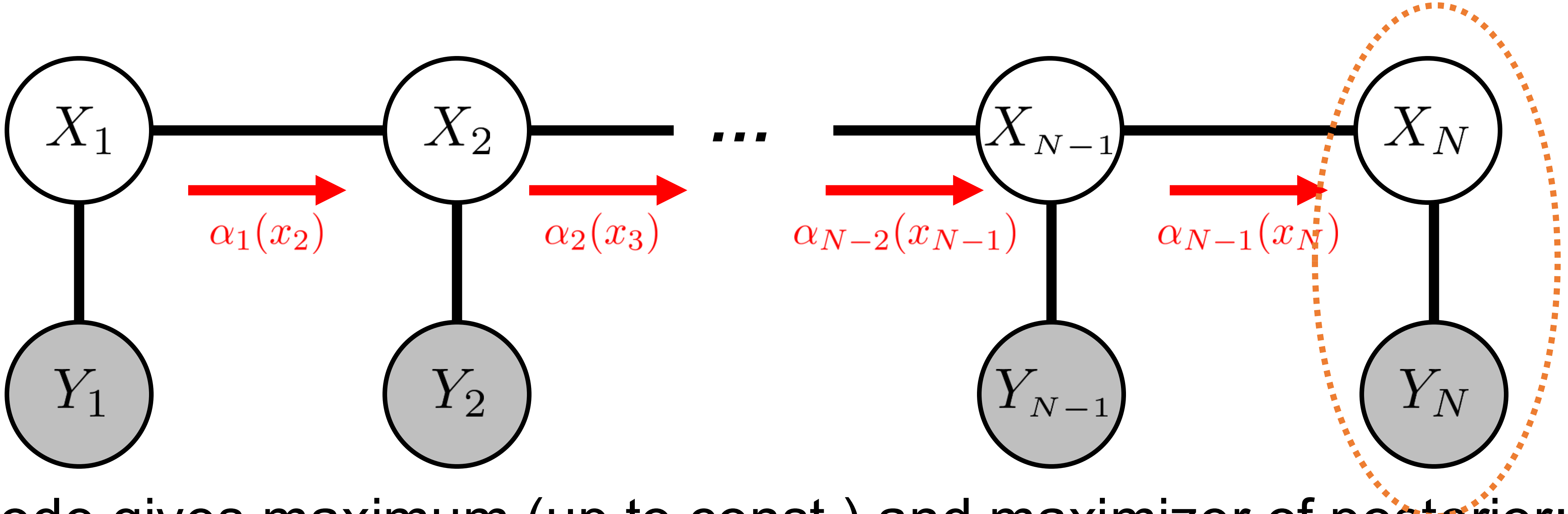
$$\alpha_{n-1}(x_n) = \max_{x_{n-1}} \log\psi(x_n, y_n) + \alpha_{n-2}(x_{n-1}) + \log\psi(x_{n-1}, x_n)$$

We also store the argmax values:

$$x^*_{n-1}(x_n) = \arg\max_{x_{n-1}} \log\psi(x_n, y_n) + \alpha_{n-2}(x_{n-1}) + \log\psi(x_{n-1}, x_n)$$

Viterbi Algorithm

Maximize instead of marginalize…



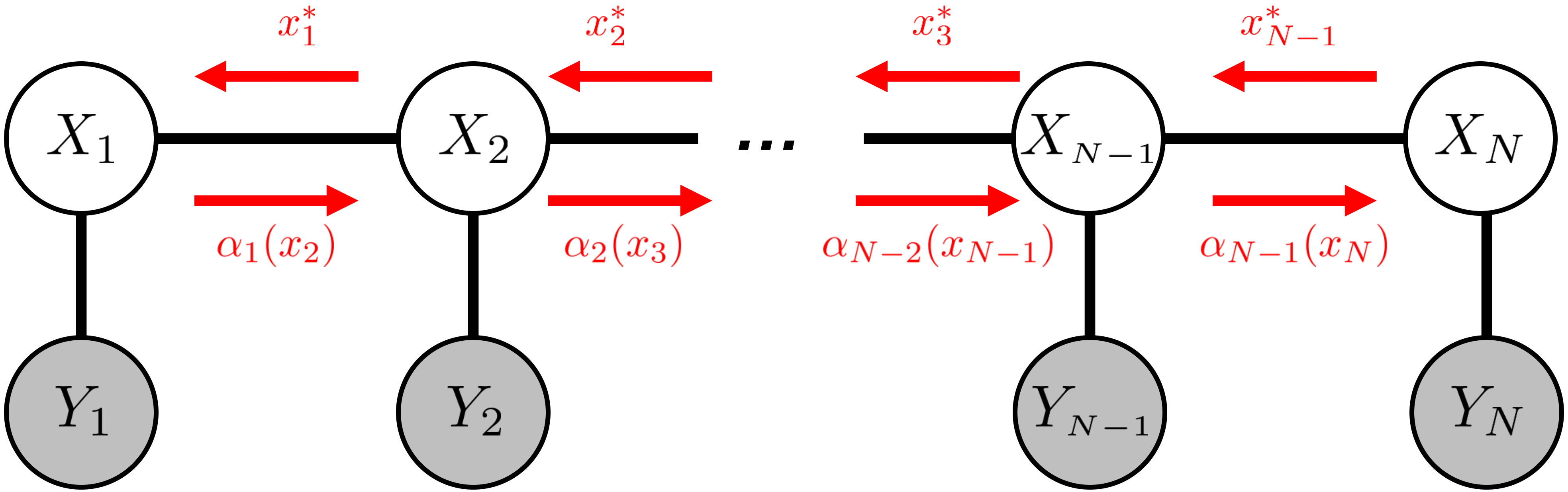
Final node gives maximum (up to const.) and maximizer of posterior:

$$\alpha_{N-1}(x_N) = \max_{x_1,\ldots,x_{N-1}} \log p(x_1, \ldots, x_N \mid \mathbf{y}) + \text{const.}$$

$$x^*_{N-1}(x_N) = \arg\max_{x_1,\ldots,x_{N-1}} \log p(x_1, \ldots, x_N \mid \mathbf{y}) + \text{const.}$$

Viterbi Algorithm

Backwards pass reads off joint maximizer…



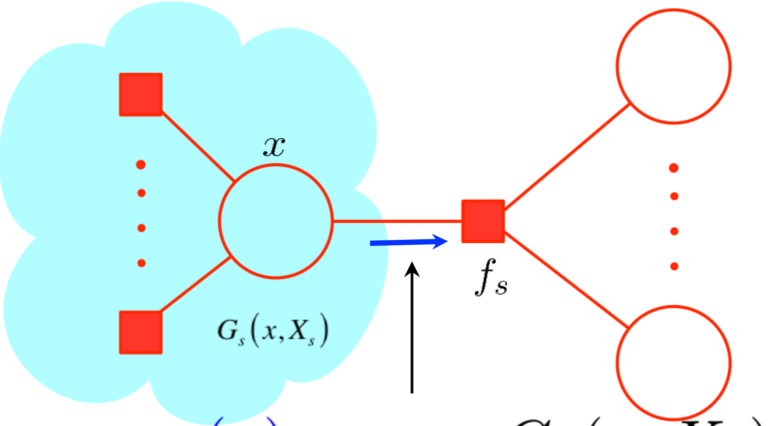
Backward Pass: $x_n^* = x_n^*(x_{n+1}^*)$

Joint maximizing sequence obtained at the end of backwards pass:

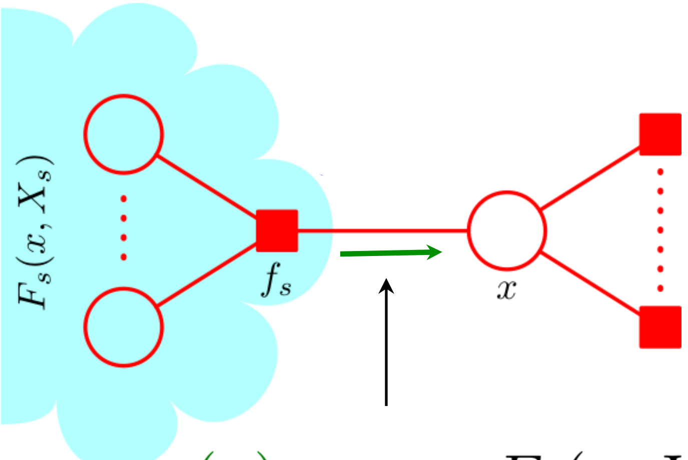
$$(x_1^*, x_2^*, \ldots, x_N^*) = \arg\max_{\mathbf{x}} p(\mathbf{x} \mid \mathbf{y})$$

Max-Product (Max-Sum) Algorithm

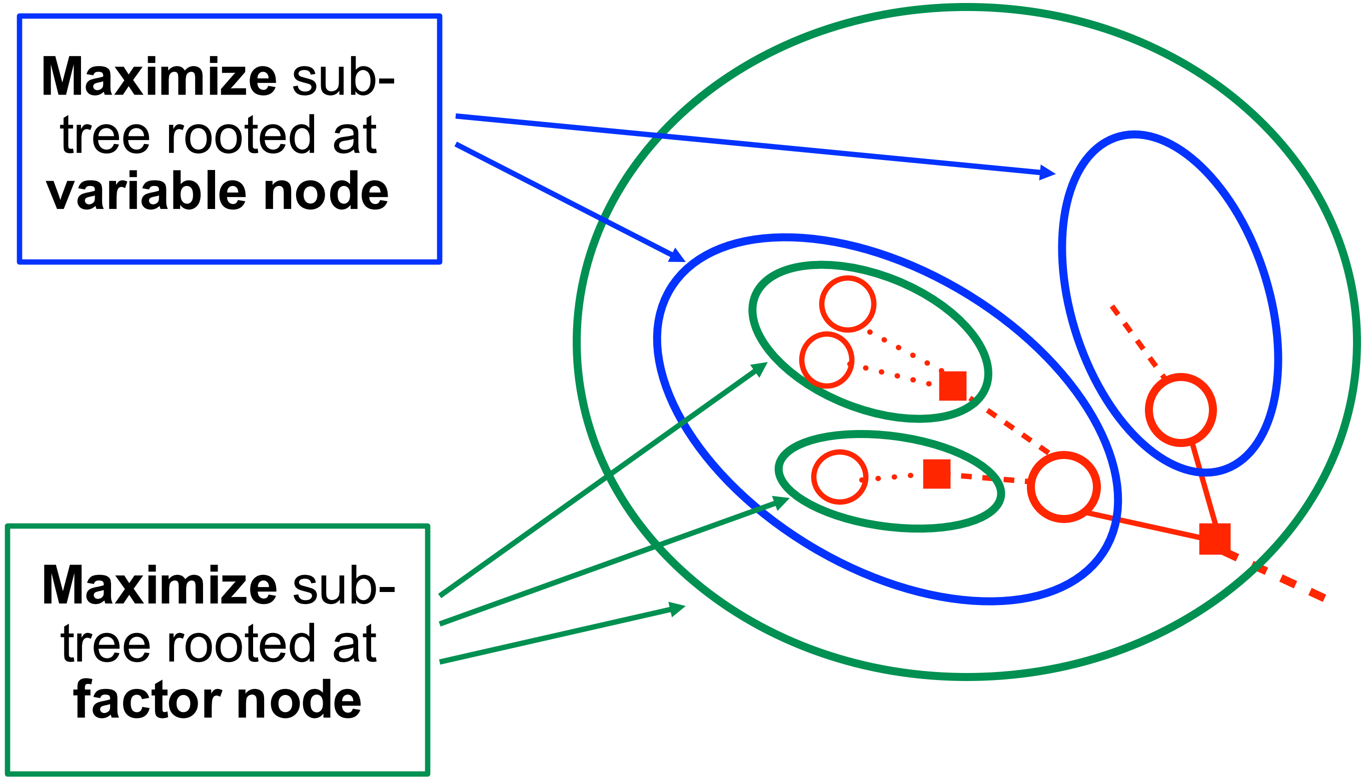
Recall our decomposition of factor graph sub-trees…



$$\mu_{x \to f_s}(x) = \max_{X_s} G_s(x, X_s)$$



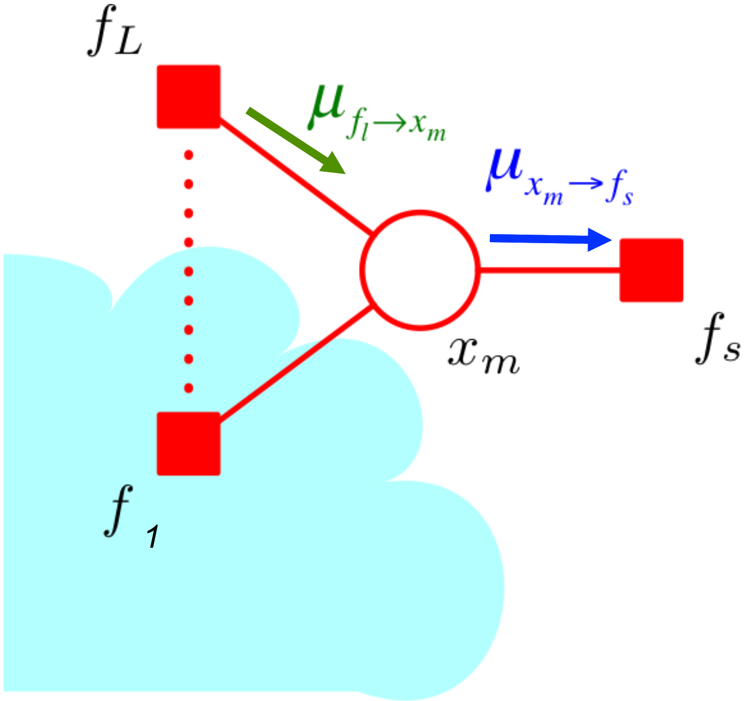
$$\mu_{f_s \to x}(x) = \max_{X_s} F_s(x, X_s)$$



Maximize sub-tree rooted at **variable node**

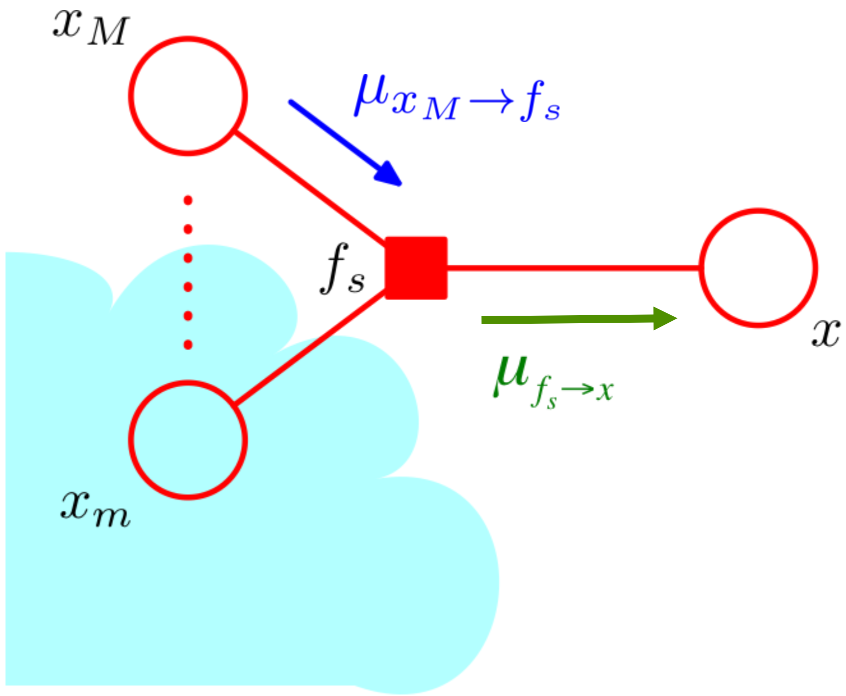
Maximize sub-tree rooted at **factor node**

Max-product on a slide



Variable x_m gathers incoming messages and sends:

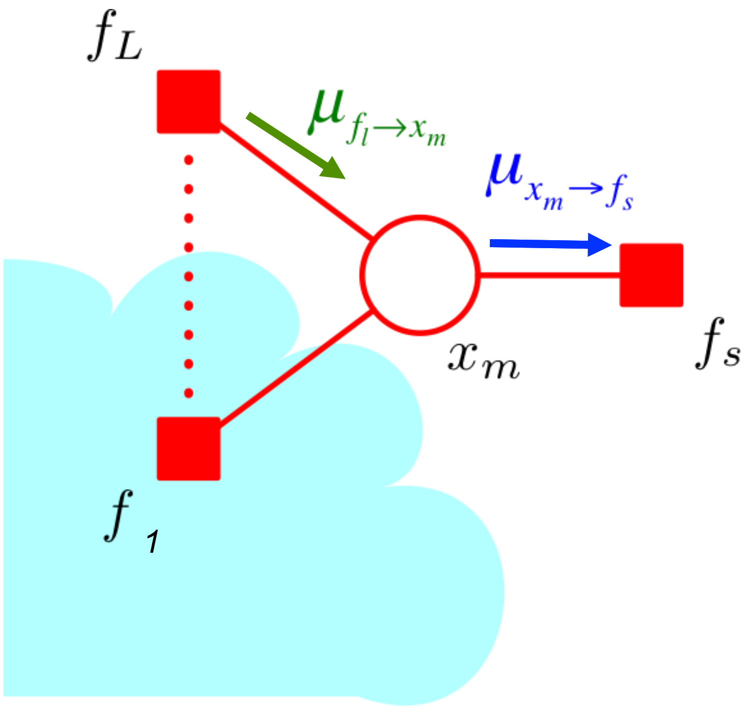
$$\mu_{x_m\to f_s(x_m)} = \prod_{f_l\in\mathrm{ne}(x_m)\backslash f_s} \mu_{f_l\to x_m}(x_m)$$

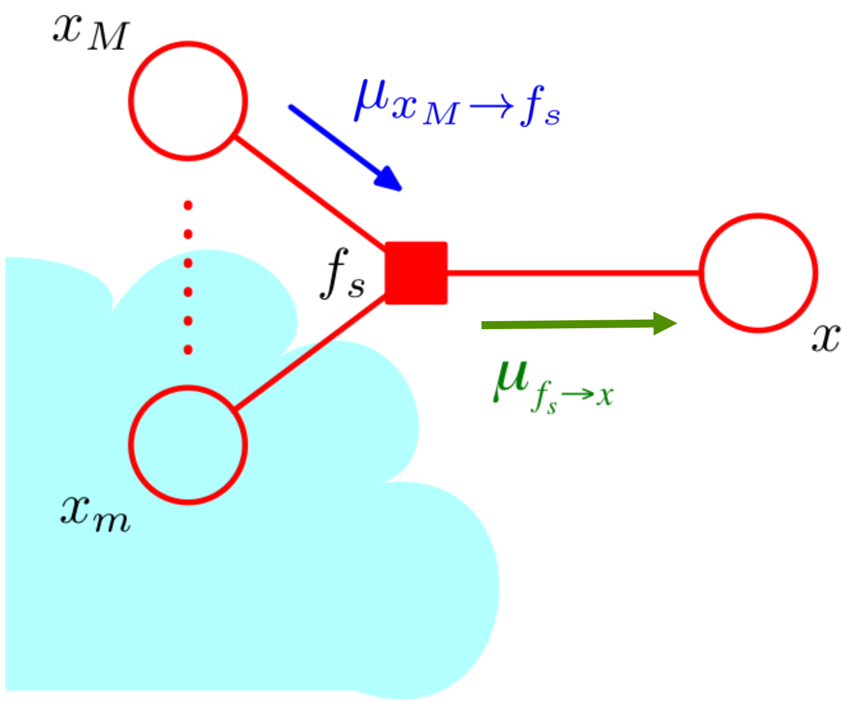


Factor f_s gathers incoming messages and sends:

$$\mu_{f_s\to x}(x) = \max_{\mathbf{x}\backslash x} f_s(x, x_1, x_2, \ldots, x_M) \prod_{m\in\mathrm{ne}(f_s)\backslash x} \mu_{x_m\to f_s}(x_m)$$

Max-sum on a slide





Variable x_m gathers incoming messages and sends:

$$\log \mu_{x_m \to f_s(x_m)} = \sum_{f_l \in \mathrm{ne}(x_m) \setminus f_s} \log \mu_{f_l \to x_m}(x_m)$$

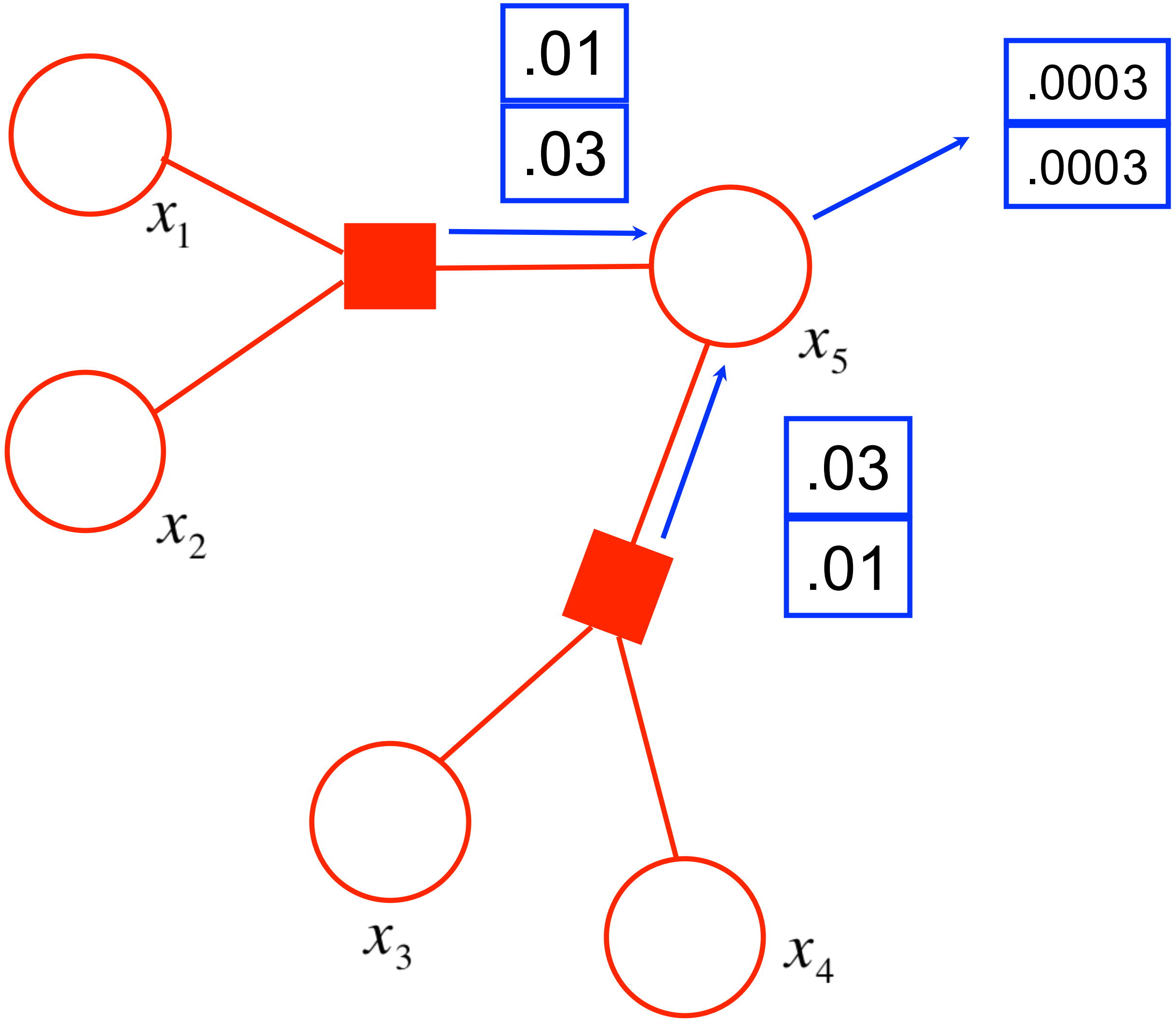
Factor f_s gathers incoming messages and sends:

$$\log \mu_{f_s \to x}(x) =$$

$$\max_{\mathbf{x} \setminus x} \log f_s(x, x_1, x_2, \ldots, x_M) + \sum_{m \in \mathrm{ne}(f_s) \setminus x} \log \mu_{x_m \to f_s}(x_m)$$

More numerically stable to work in log-domain (max-sum)…

Max-sum Example



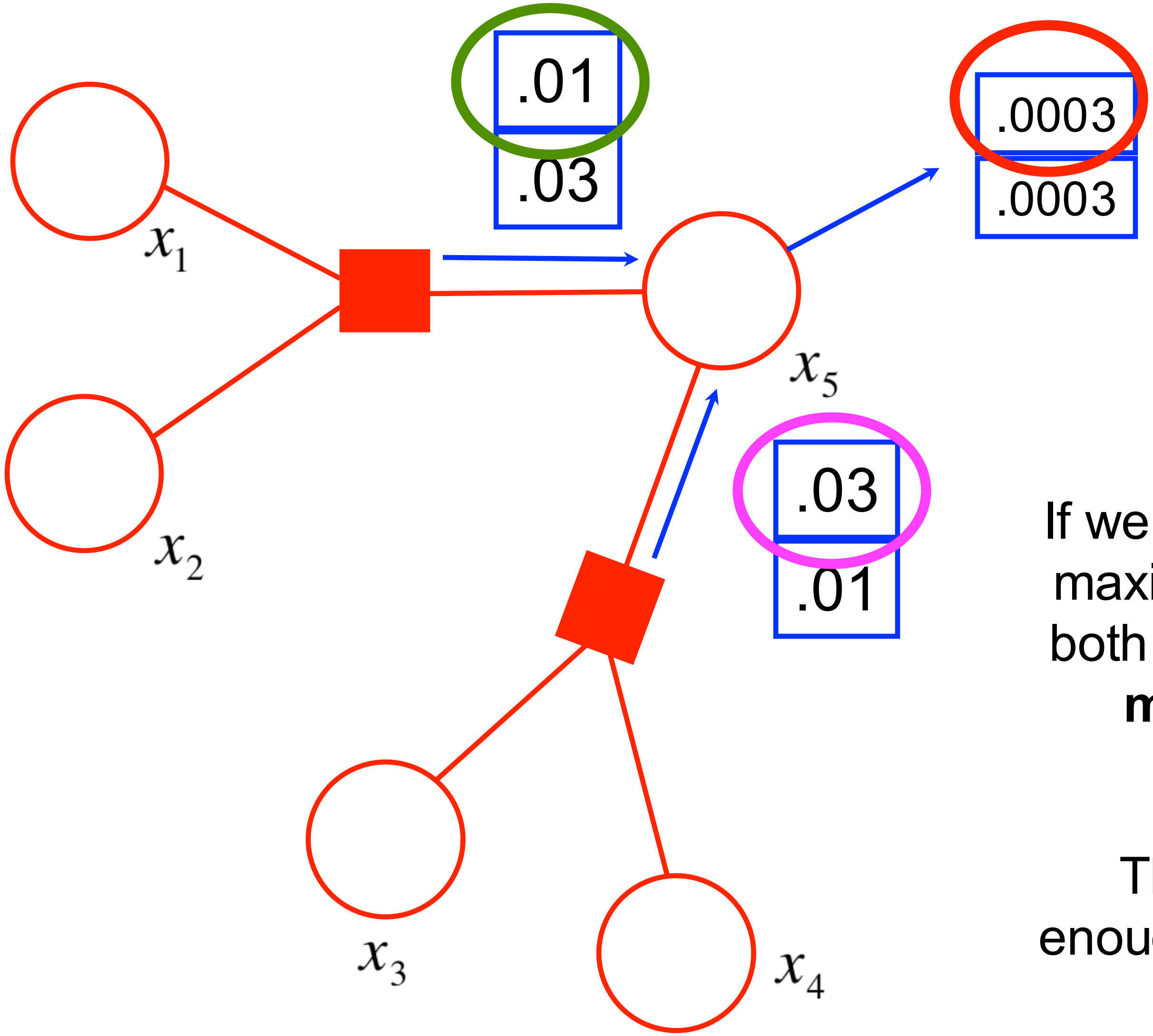
Now we know that
x_5 can either be 0 or
1 (it is a tie)

Max-sum Example

- At the root we can record the argmax for its variable, but we do not know which variable choices produced it
  - Ties have the potential to make this particularly complicated

- We can “backtrack” to find this out provided that we stored what we need in the forward pass.

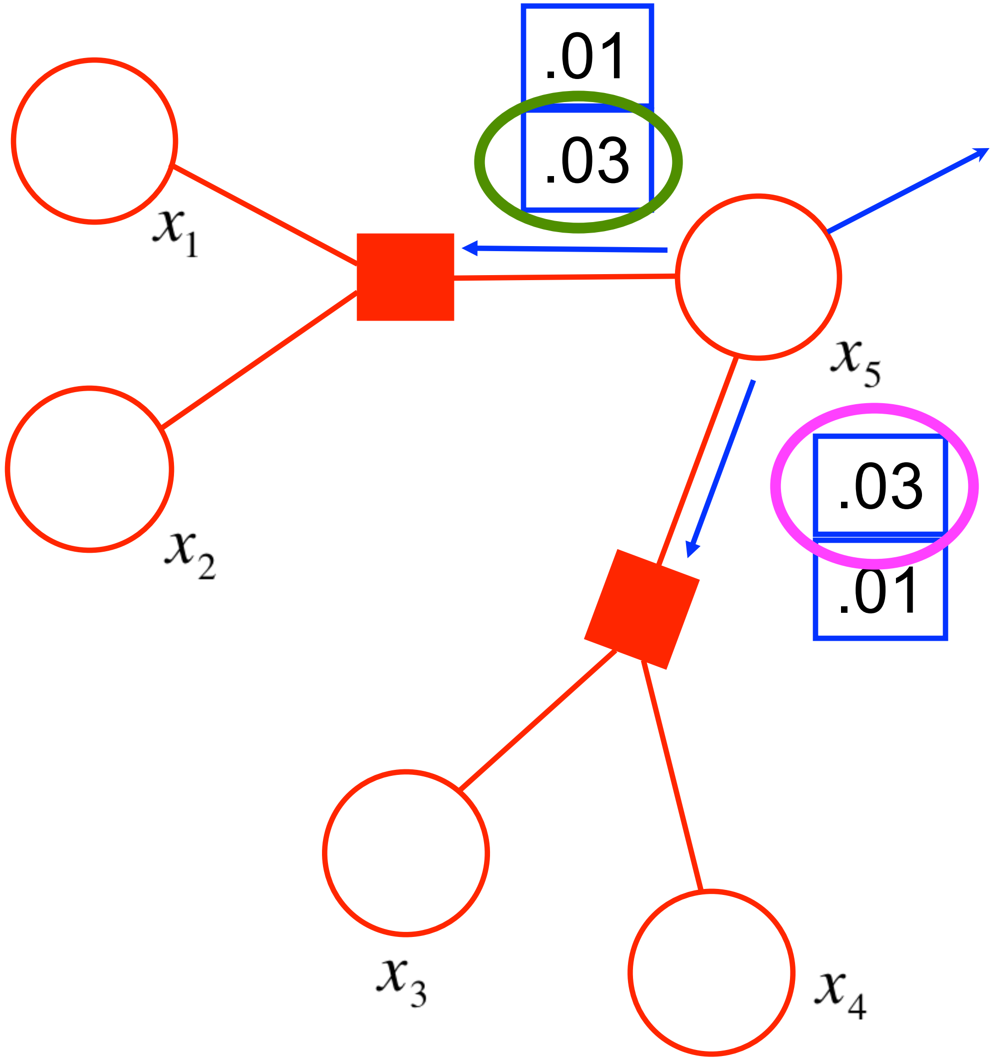
- If there are ties, they need to be handled consistently
  - In our example, we need to choose either $x_5 = 0$ **or** $x_5 = 1$ for **both** backtracking branches.



If we choose x_5=0, then we need a maximal configuration for x_5=0 for both pieces for a **consistent joint maximizing configuration**

The factor nodes must store enough information to evaluate any choice

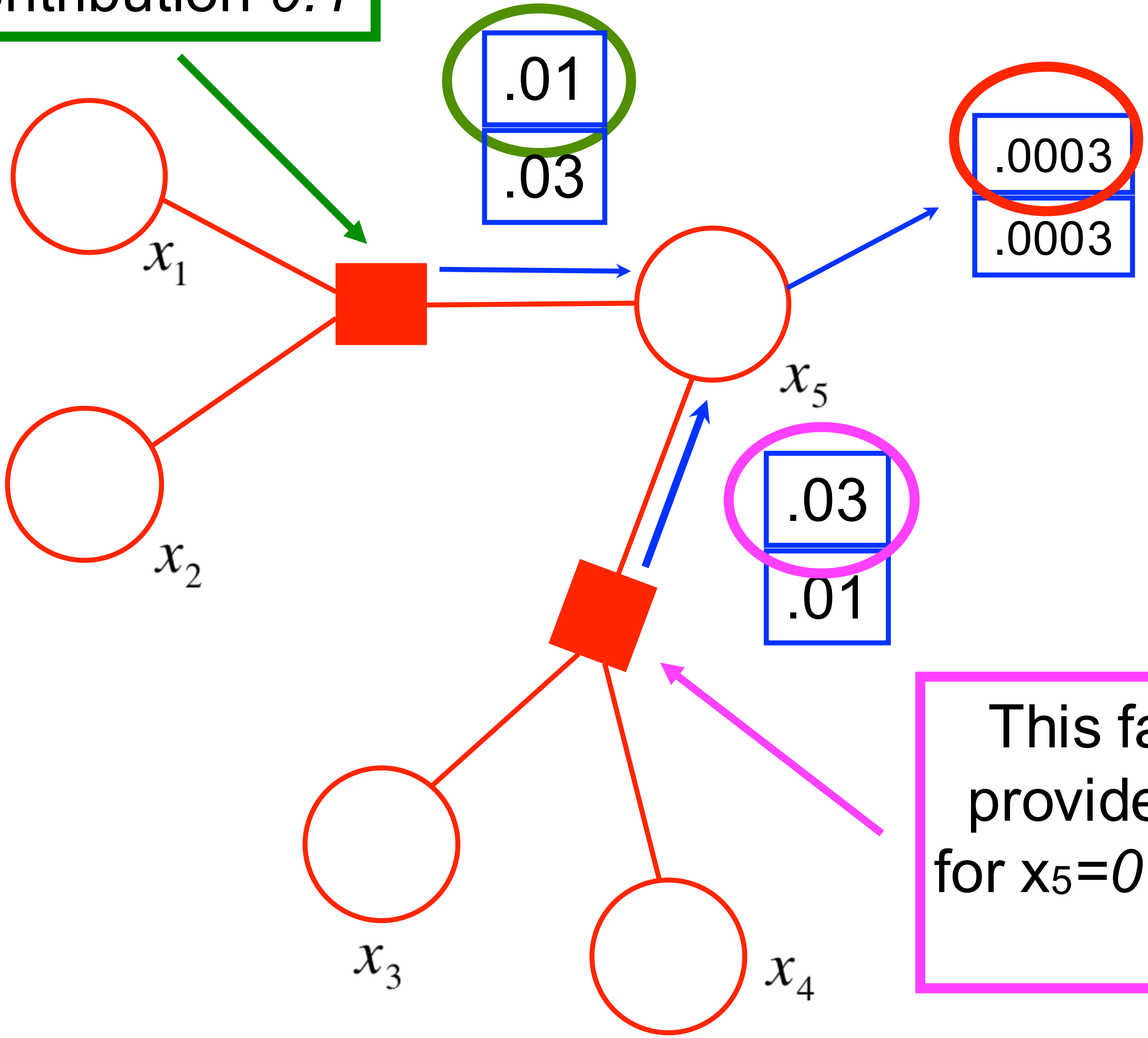
WRONG



The configuration that we get backtracking pretending $x_5 = 0$, even though $x_5 = 1$cannot compute to more than 0.1, and could be less, as the settings for the other variables are making the value as big as possible when $x_5 = 0$.

For $x_5=0$

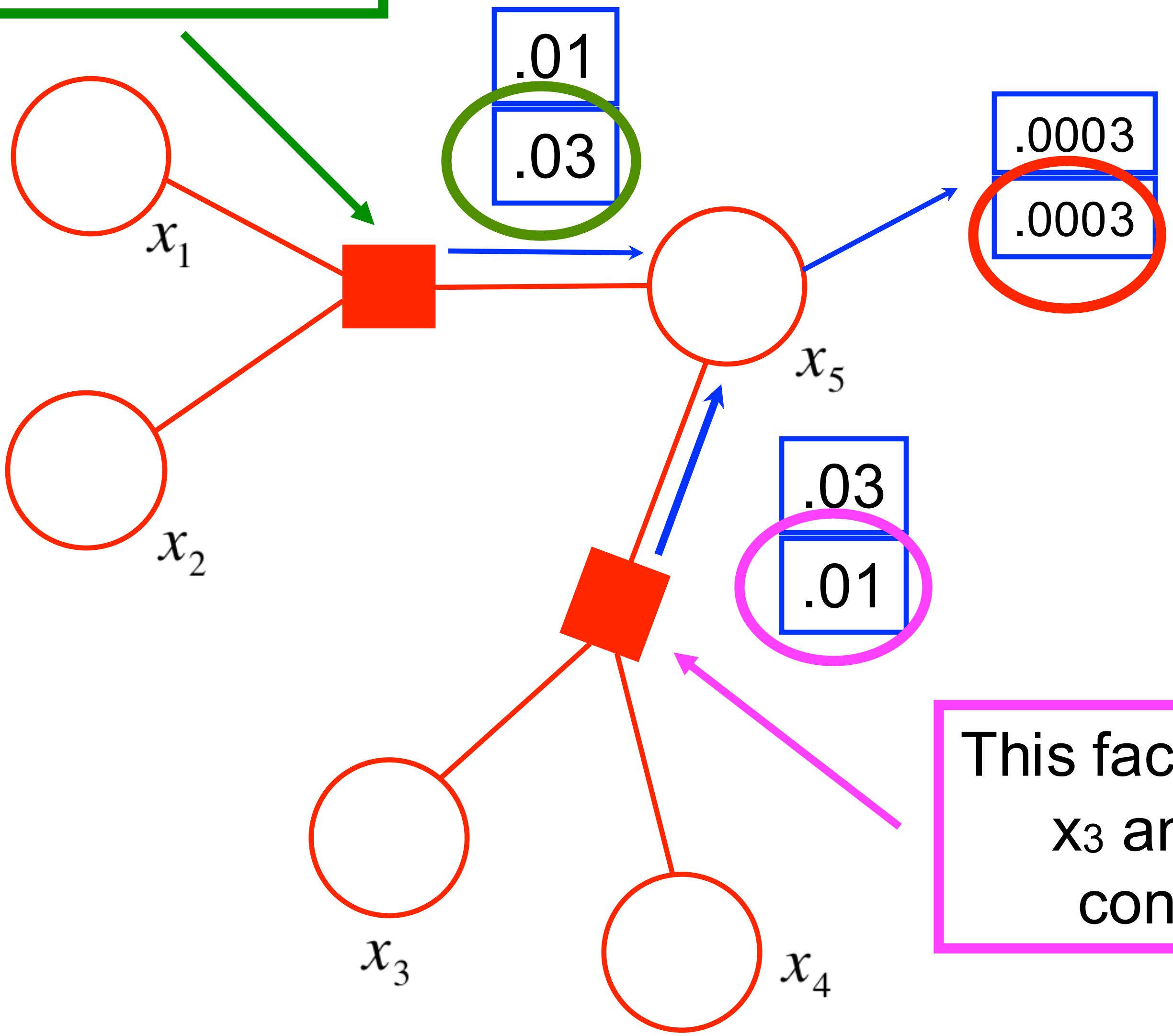
This factor must provide x_1 and x_2 for $x_5=0$ contribution *0.1*



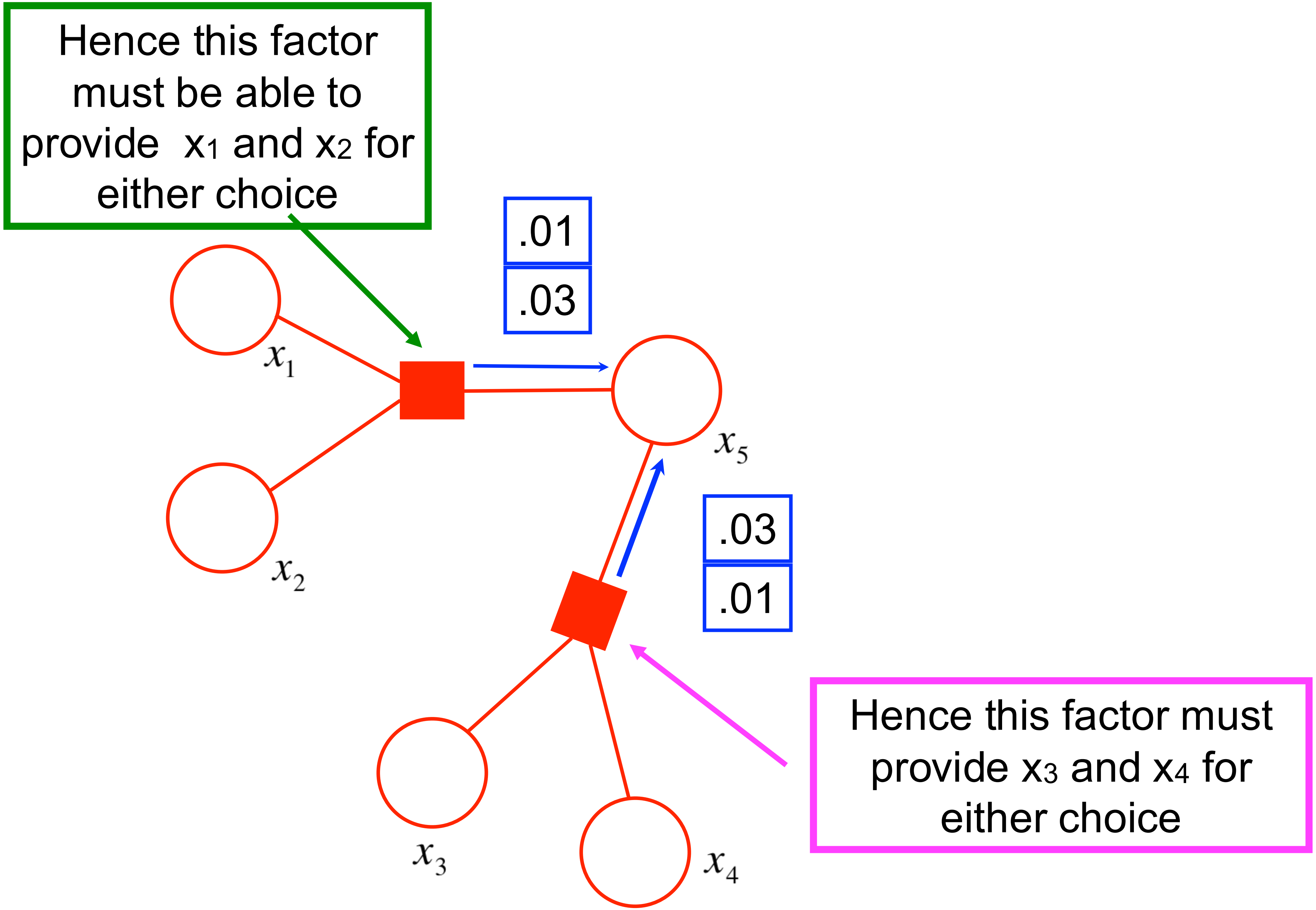
This factor must provide x_3 and x_4 for $x_5=0$ contribution *0.3*

For x_5=*1*

This factor must provide x_1 and x_2 for x_5=*1* contribution *0.3*



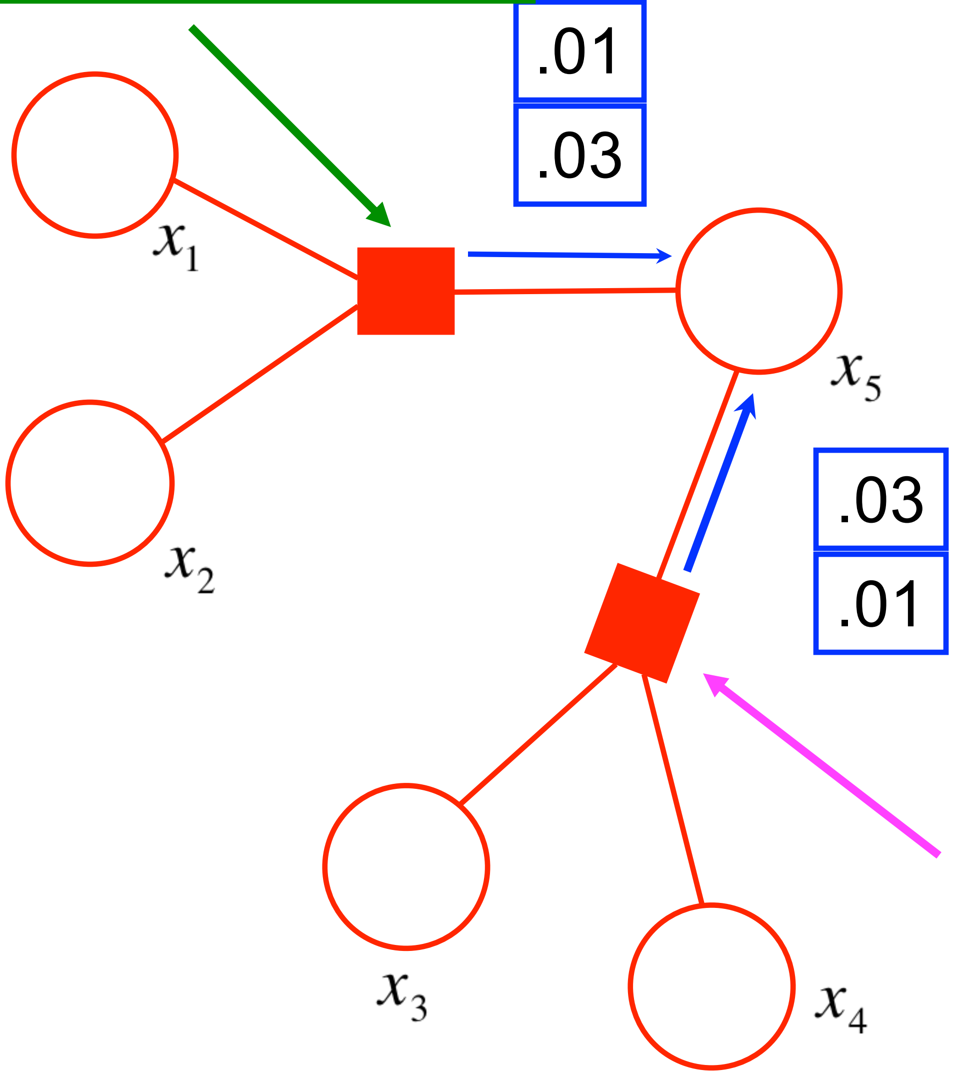
This factor must provide x_3 and x_4 for x_5=*1* contribution *0.1*



Hence this factor must be able to provide x_1 and x_2 for either choice

Hence this factor must provide x_3 and x_4 for either choice

Store $\arg\max_{x_1,x_2} \left\{ \left(f_A \rightarrow x_5 \right)_i \right\}$



Store

$\arg\max_{x_3,x_4} \left\{ \left(f_B \rightarrow x_5 \right)_i \right\}$

Message Passing Inference Summary

- Brute-force enumeration exponential regardless of graph
- Sum-Product BP
  - Exact inference in tree-structure graphs in O(TK^2) time for T nodes, each taking K states
  - Reduces to Forward-Backward in HMMs
  - Same for Max-Product BP (reduces to Viterbi in HMMs)
- Variable elimination
  - Exact marginals in general graphs
  - Worst-case complexity exponential in size of largest clique
  - Need to rerun from scratch for each marginal
  - Complexity dependent on elimination order (NP-hard to optimize)

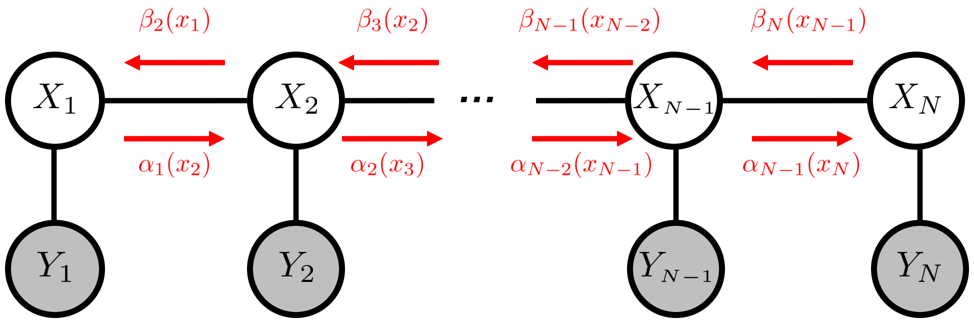
Message Passing Inference Summary

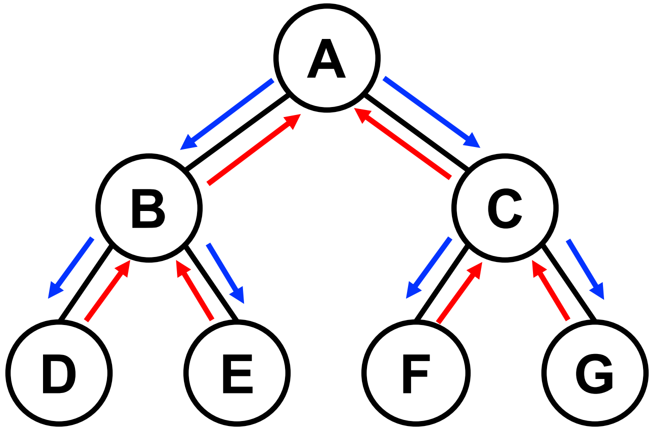
- Junction Tree Algorithm
  - Exact marginals in general graphs
  - Caches messages to compute all marginals
  - Worst-case complexity exponential in size of largest clique
  - Optimizing Jtree is NP-hard (corresponds to finding treewidth)

- Loopy BP: Just did this, did you forget already?

Message Passing Inference Summary

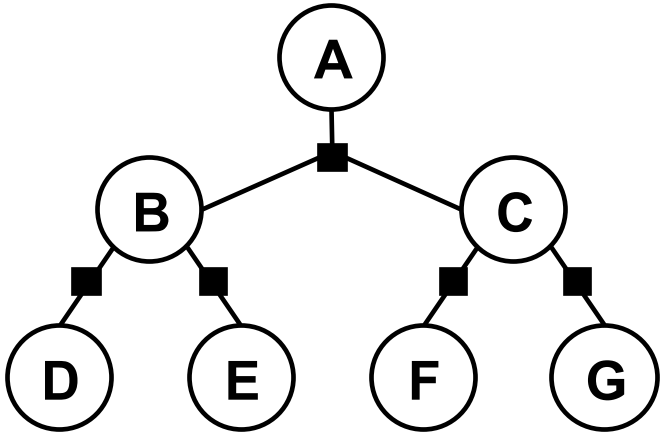
Forward-backward algorithm yields efficient marginal inference on HMM graph





Sum-product belief propagation generalizes marginal inference to tree-structured MRFs

Max-product / max-sum yields maximum a posteriori (MAP) inference in any tree-structured model



Viterbi decoder is special case for HMM

And factor graphs