Probabilistic and Bayesian Machine Learning

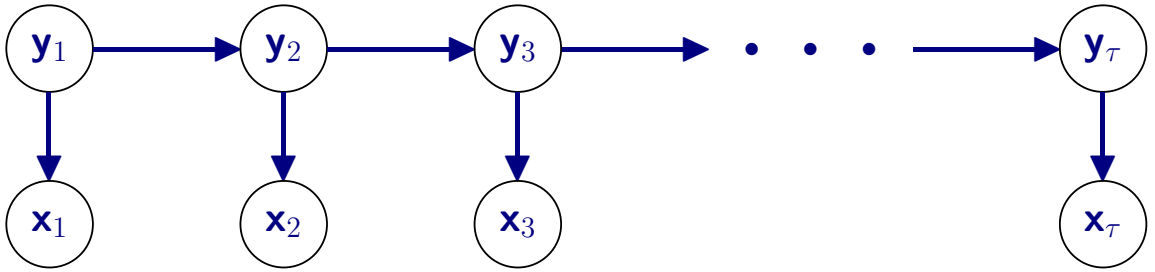
Day 3: The Junction Tree Algorithm

Yee Whye Teh
ywteh@gatsby.ucl.ac.uk

Gatsby Computational Neuroscience Unit
University College London

http://www.gatsby.ucl.ac.uk/∼ywteh/teaching/probmodels

Inference for HMMs



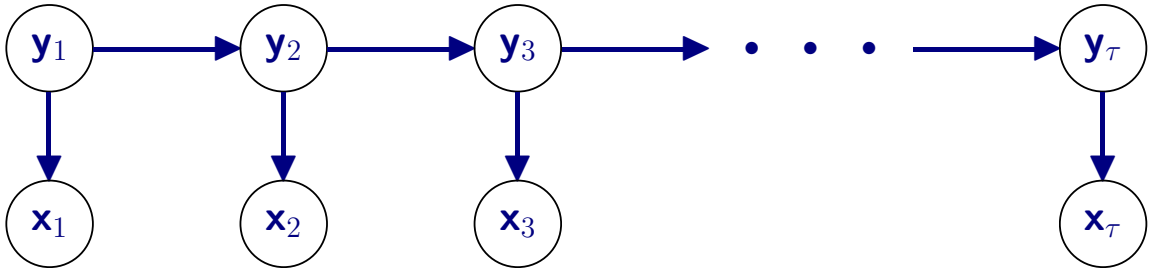
$$P(Y_{1:\tau}, X_{1:\tau}) = P(Y_1)P(X_1|Y_1)\prod_{t=2}^{\tau} P(Y_t|Y_{t-1})P(X_t|Y_t)$$

Note that we suppressed writing dependence on parameters θ for clarity.

Inference and Learning Problems:

- Filtering: $P(Y_t|X_1, \ldots, X_t)$
- Prediction: $P(Y_t|X_1, \ldots, X_{t-1})$
- Smoothing: $P(Y_t|X_1, \ldots, X_\tau)$ and $P(Y_t, Y_{t+1}|X_1, \ldots, X_\tau)$
- Likelihood: $P(X_1, \ldots, X_\tau)$
- Learning: $\theta^{\mathsf{ML}} = \mathsf{argmax}_\theta\, P(X_1, \ldots, X_\tau|\theta)$

Likelihood of HMMs



The likelihood of a HMM looks to be an intractable sum of an exponential number of terms:

$$P(X_{1:\tau}) = \sum_{Y_{1:\tau}} P(Y_{1:\tau}, X_{1:\tau})$$

Fortunately, there is a forward recursion allowing us to compute the likelihood efficiently.
For $t = 1$,

$$\alpha_1(Y_1) := P(X_1, Y_1) = P(Y_1)P(X_1|Y_1)$$

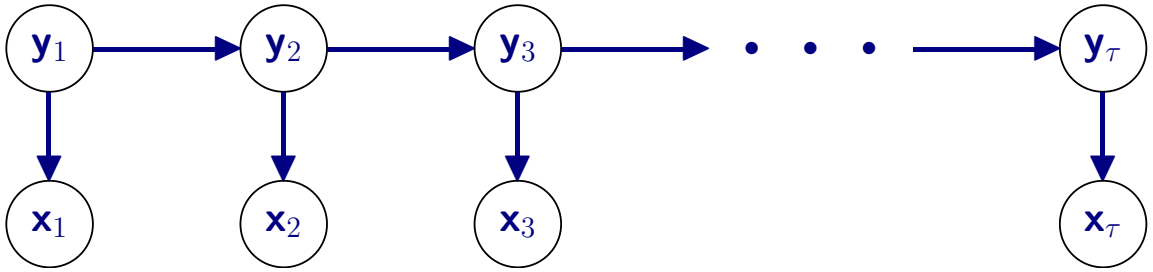
For $t = 2, \ldots, \tau$,

$$\alpha_t(Y_t) := P(X_{1:t}, Y_t) = \sum_{Y_{t-1}} \alpha_{t-1}(Y_{t-1})P(Y_t|Y_{t-1})P(X_t|Y_t)$$

The likelihood is then,

$$P(X_{1:\tau}) = \sum_{Y_\tau} P(X_{1:\tau}, Y_\tau) = \sum_{Y_\tau} \alpha_\tau(Y_\tau)$$

Filtering and Prediction of HMMs



For discrete latent variables Y_t's with K states, computational cost is $O(\tau K^2)$ instead of $O(K^\tau)$.

The forward messages $\alpha_t(y)$ also give us the filtering probabilities once normalized:

$$P(Y_t|X_{1:t}) = \alpha_t(Y_t)/\sum_{m=1}^{K}\alpha_t(m)$$

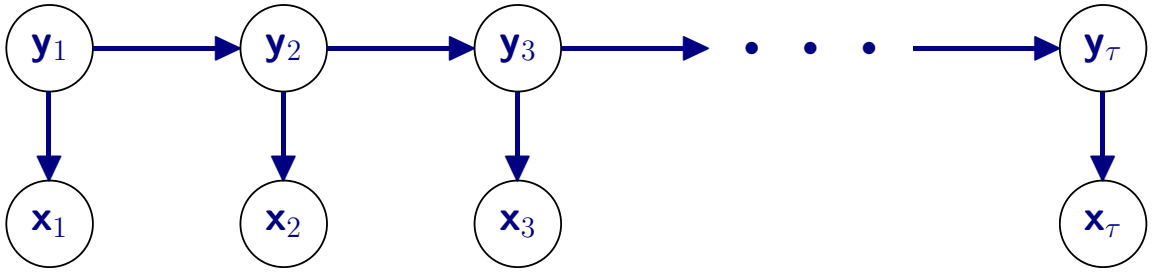
Numerical stability: in practice it is important to normalize the forward messages after each step, as each $P(X_{1:t}, Y_t)$ can become exponentially small in t:

$$Z_t = \sum_{Y_t}\alpha_t(Y_t) \qquad\qquad \alpha'_t(Y_t) := \alpha_t(Y_t)/Z_t$$

The log likelihood is then the sum of the normalization constants:

$$\log P(X_{1:\tau}) = \sum_{t=1}^{\tau}\log Z_t$$

Smoothing of HMMs



For smoothing, we also require to incorporate information from observations at later times. This requires computation of backward messages which carry this information.
For $t = \tau$,

$$\beta_\tau(Y_\tau) := 1$$

For $t = \tau - 1, \ldots, 1$,

$$\beta_t(Y_t) = P(X_{t+1:\tau}|Y_t) = \sum_{Y_{t+1}} P(Y_{t+1}|Y_t)P(X_{t+1}|Y_{t+1})\beta_{t+1}(Y_{t+1})$$

The marginal probabilities are then:

$$P(Y_t|X_{1:\tau}) \propto P(X_{1:t}, Y_t)P(X_{t+1:\tau}|Y_t) = \alpha_t(Y_t)\beta_t(Y_t)$$

And the pairwise probabilities are:

$$P(Y_t, Y_{t+1}|X_{1:\tau}) \propto \alpha_t(Y_t)P(Y_{t+1}|Y_t)P(X_{t+1}|Y_{t+1})\beta_{t+1}(Y_{t+1})$$

The algorithm is variously known as the forward-backward algorithm, the sum-product algorithm, or belief propagation.

Distributive Law

The belief propagation algorithm can be understood as implementing the distributive law of multiplication over addition. In particular the backward messages are given by:

$$
\begin{aligned}
P(X_{1:\tau}) &= \sum_{Y_{1:\tau}} P(Y_{1:\tau}, X_{1:\tau}) \\
&= \sum_{Y_{1:\tau}} P(Y_1)P(X_1|Y_1) \prod_{t=2}^{\tau} P(Y_t|Y_{t-1})P(X_t|Y_t) \\
&= \sum_{Y_1} P(Y_1)P(X_1|Y_1) \sum_{Y_2} P(Y_2|Y_1)P(X_2|Y_2) \sum_{Y_3} \cdots \sum_{Y_\tau} P(Y_\tau|Y_{\tau-1})P(X_\tau|Y_\tau)
\end{aligned}
$$

Similarly the forward messages can be obtained by the reversed way of distributing the sums into the products.

Other belief propagation algorithms can be obtained with different distributive laws:

- Max-product: multiplications distribute over maximizations as well. The max-product algorithm computes the most likely state trajectory:

$$
Y_{1:\tau}^{\mathsf{MAP}} = \operatorname*{argmax}_{Y_{1:\tau}} P(Y_{1:\tau}, X_{1:\tau})
$$

- Max-sum: Take logarithms before applying max-product gives the max-sum. This is more efficient (addition computations are faster than multiplications on CPUs) and numerically stable.

Undirected Graphs and Partition Functions

$$P(X_{1:\tau}) = \sum_{Y_{1:\tau}} P(Y_1)P(X_1|Y_1)\prod_{t=2}^{\tau} P(Y_t|Y_{t-1})P(X_t|Y_t)$$

Define factors for an undirected graphical model over the latent variables $Y_{1:\tau}$ only:

$$f_t(Y_t) = P(X_t|Y_t) \qquad\qquad f_{t,t+1}(Y_t, Y_{t+1}) = P(Y_{t+1}|Y_t)$$

The joint distribution given by the undirected graph is:

$$P^U(Y_{1:\tau}) = \frac{1}{Z}\prod_t f_t(Y_t)f_{t,t+1}(Y_t, Y_{t+1})$$

The normalization constant (also called the partition function) of the distribution is precisely:

$$P(X_{1:\tau}) = Z = \sum_{Y_{1:\tau}}\prod_t f_t(Y_t)f_{t,t+1}(Y_t, Y_{t+1})$$

While the smoothed marginals are just the marginal distributions of P^U:

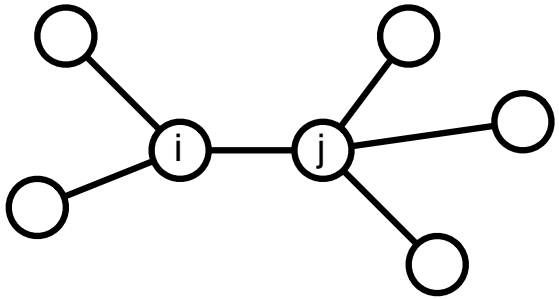
$$P(Y_t|X_{1:\tau}) = \frac{P(Y_t, X_{1:\tau})}{P(X_{1:\tau})} = \frac{\sum_{Y_{1:t-1,t+1:\tau}}\prod_{t'} f_{t'}(Y_{t'})f_{t',t'+1}(Y_{t'}, Y_{t'+1})}{Z} = P^U(Y_t)$$

$$P(Y_t, Y_{t+1}|X_{1:\tau}) = P^U(Y_t, Y_{t+1})$$

Belief Propagation on Undirected Trees

Undirected tree parameterization for joint distribution:

$$p(\mathbf{Y}) = \frac{1}{Z} \prod_{\text{edges } (ij)} f_{(ij)}(Y_i, Y_j)$$



Define $T_{i \to j}$ to be the subtree containing i if j is removed. Define **messages**:

$$M_{i \to j}(Y_j) = \sum_{Y_{T_{i \to j}}} f_{(ij)}(Y_i, Y_j) \prod_{\text{edges } (i'j') \text{ in } T_{i \to j}} f_{(i'j')}(Y_{i'}, Y_{j'})$$

Then the messages can be recursively computed as follows:

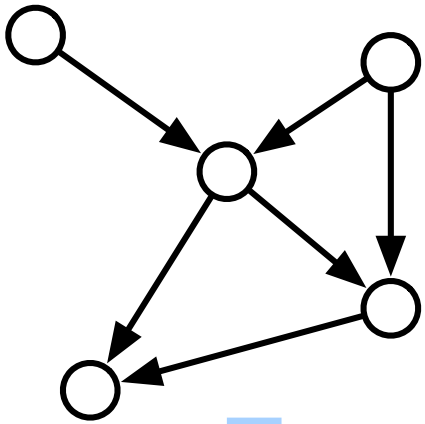
$$M_{i \to j}(Y_j) = \sum_{Y_i} f_{(ij)}(Y_i, Y_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \to i}(Y_i)$$

and:

$$p(Y_i) \propto \prod_{k \in \text{ne}(i)} M_{k \to i}(Y_i)$$

$$p(Y_i, Y_j) \propto f_{(ij)}(Y_i, Y_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \to i}(Y_i) \prod_{l \in \text{ne}(j) \setminus i} M_{l \to j}(Y_j)$$

Recap on Graphical Models



$$\{X_i \perp\!\!\!\perp Y_i | \mathcal{V}_i\}$$

$$\{p(\mathbf{X}) = \textstyle\prod_i p(X_i | X_{\mathsf{pa}(i)})\}$$

$$\{p(\mathbf{X}) : p(X_i, Y_i | \mathcal{V}_i) = p(X_i | \mathcal{V}_i) p(Y_i | \mathcal{V}_i)\}$$

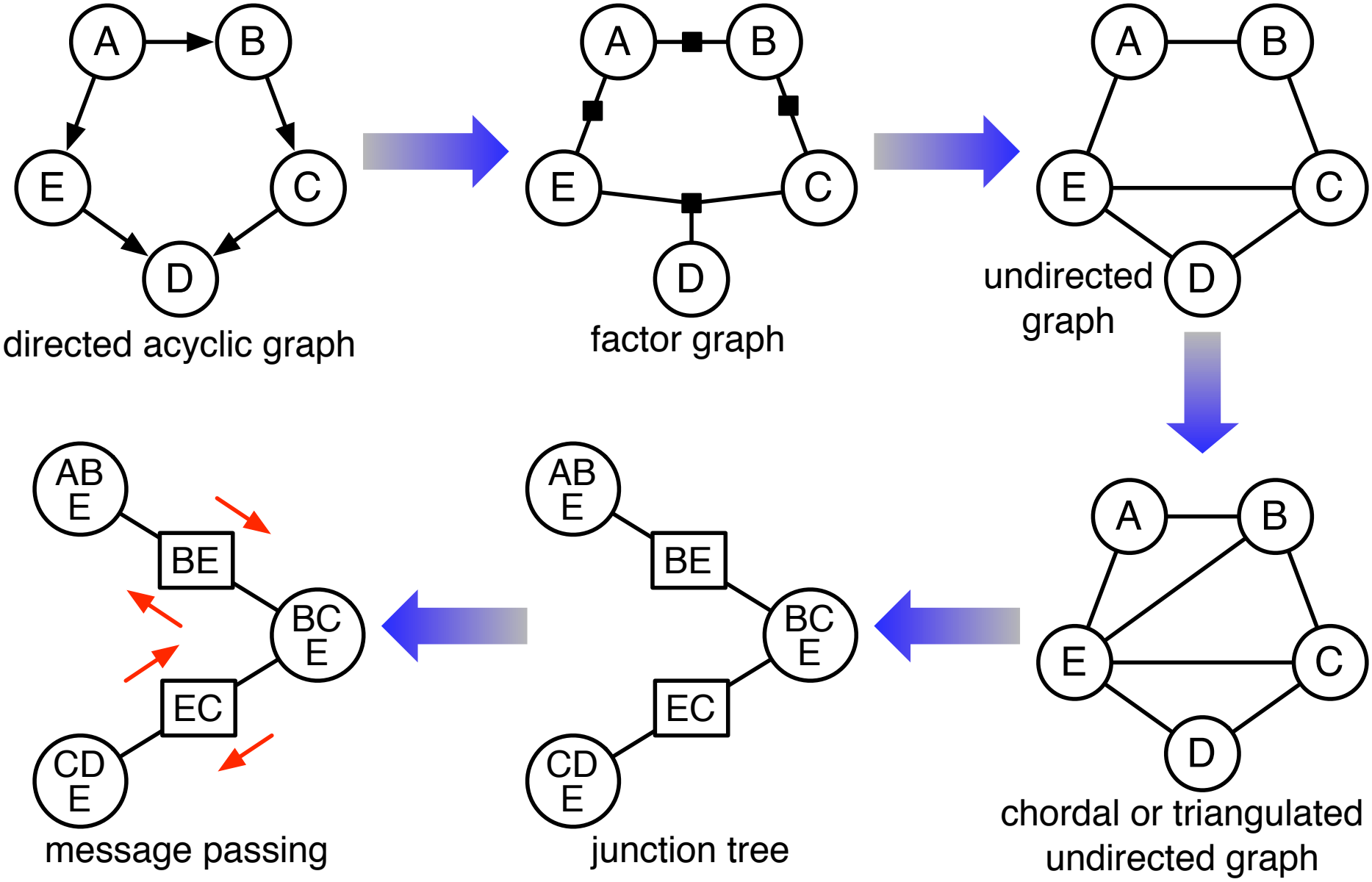
Graph Transformations

Our general approach to inference in arbitrary graphical models is to transform these graphical models to ones belonging to an easy-to-handle class (specifically, **junction or join trees**).

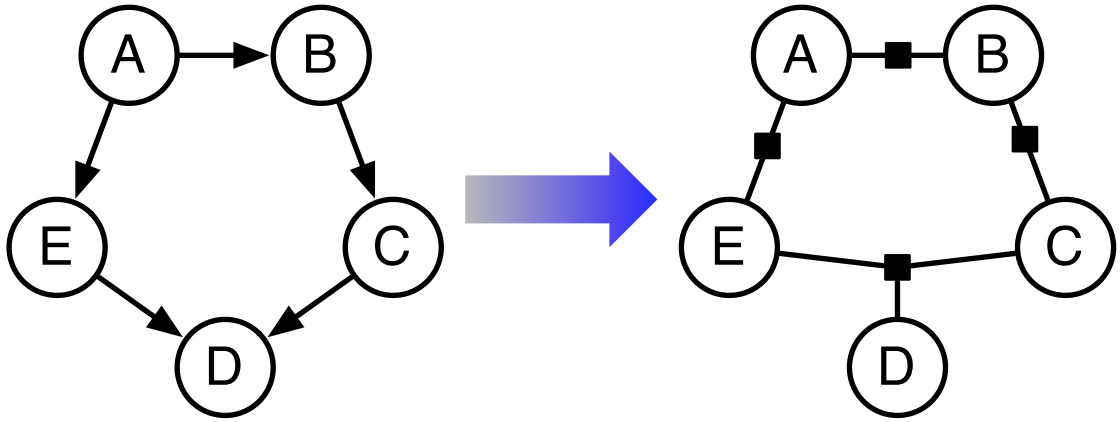
Suppose we are interested in inference in a distribution $p(\mathbf{Y})$ representable in a graphical model G.

- In transforming G to an easy-to-handle G', we need to ensure that $p(\mathbf{Y})$ is representable in G' too.
- This can be ensured by making sure that every step of the graph transformation only removes conditional independencies, never adds them.
- This guarantees that the family of distributions can only grow at each step, and $p(\mathbf{Y})$ will be in the family of distributions represented by G'.
- Thus inference algorithms working on G' will work for $p(\mathbf{Y})$ too.

The Junction Tree Algorithm



Directed Acyclic Graphs to Factor Graphs



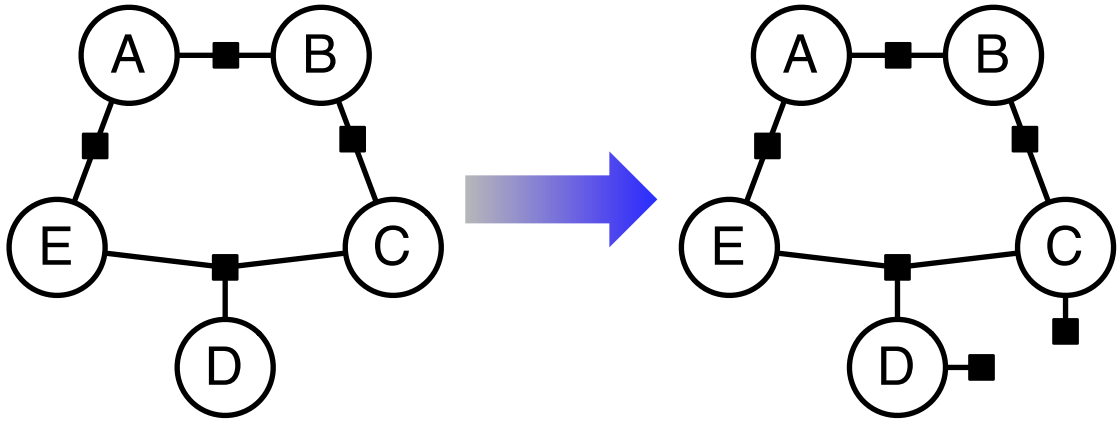
Factors are simply the conditional distributions in the DAG.

$$p(\mathbf{Y}) = \prod_i p(Y_i|Y_{\mathsf{pa}(i)})$$
$$= \prod_i f_i(Y_{C_i})$$

where $C_i = i \cup \mathsf{pa}(i)$ and $f_i(Y_{C_i}) = p(Y_i|Y_{\mathsf{pa}(i)})$.

Marginal distribution on roots $p(Y_r)$ absorbed into an adjacent factor.

Entering Evidence in Factor Graphs



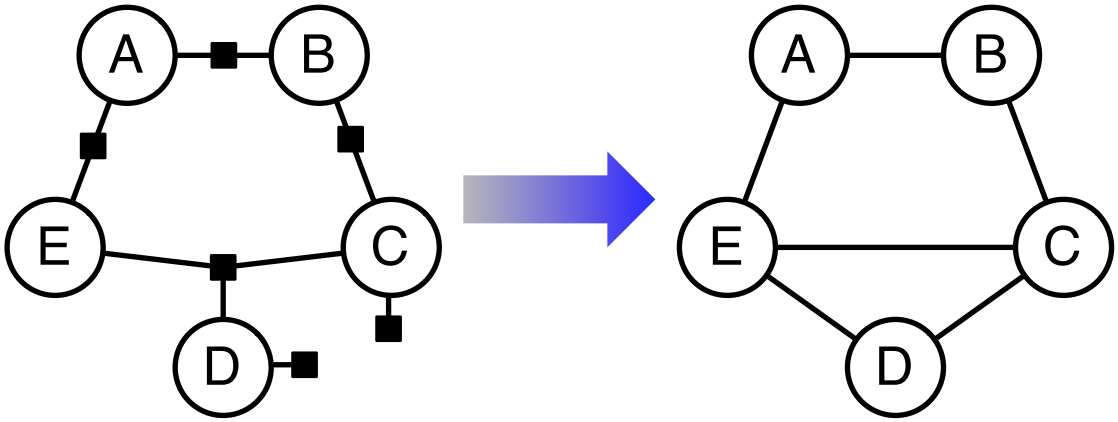
Often times we are interested in inferring posterior distributions given observed evidence (e.g. $D =$ wet and $C =$ rain).

This can be achieved by adding factors with just one adjacent node, with

$$f_D(D) = \begin{cases} 1 & \text{if } D = \text{wet;} \\ 0 & \text{otherwise.} \end{cases}$$

$$f_C(C) = \begin{cases} 1 & \text{if } C = \text{rain;} \\ 0 & \text{otherwise.} \end{cases}$$

Factor Graphs to Undirected Graphs



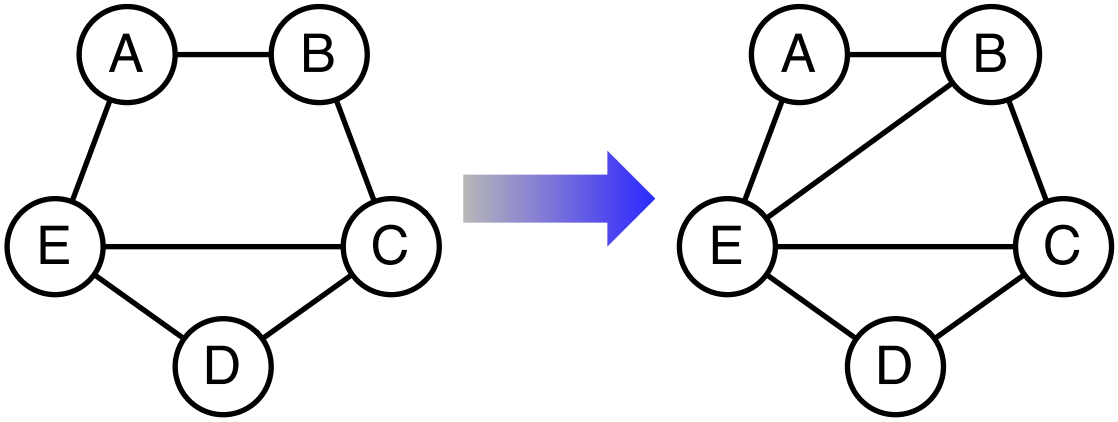
Just need to make sure that every factor is contained in some maximal clique of the undirected graph.

$$p(\mathbf{Y}) = \frac{1}{Z} \prod_i f_i(Y_{C_i})$$

We can make sure of this simply by converting each factor into a clique, and absorbing $f_i(Y_{C_i})$ into the factor of some maximal clique containing it.

The transformation DAG $\Rightarrow$ undirected graph is called **moralization**—we simply "marry" all parents of each node by adding edges connecting them, then drop all arrows on edges.

Triangulation of Undirected Graphs



Message passing—messages contain information from other parts of the graph, and this information is propagated around the graph during inference.

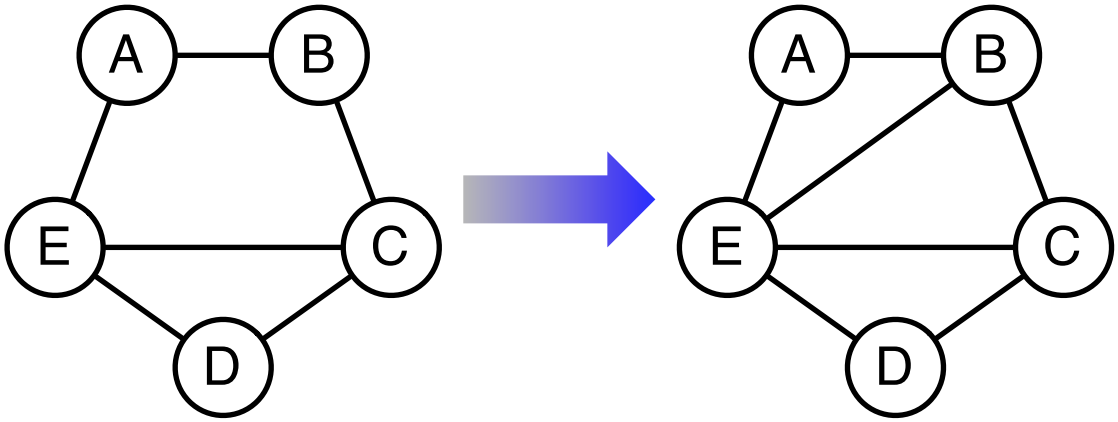
If loops (cycles) are present in the graph, message passing can lead to overconfidence due to double counting of information, and to oscillatory (non-convergent) behaviour[1].

To prevent this overconfident and oscillatory behaviour, we need to make sure that different channels of communication coordinate with each other to prevent double counting.

Triangulation: add edges to the graph so that every loop of size > 4 has at least one chord. Note recursive nature: adding edges often creates new loops; we need to make sure new loops of length > 4 have chords too.

[1] This is called **loopy belief propagation**, and we will see in the second half of the course that this is an important class of approximate inference algorithms.

Triangulation of Undirected Graphs



Triangulation: add edges to the graph so that every loop of size > 4 has at least one chord.

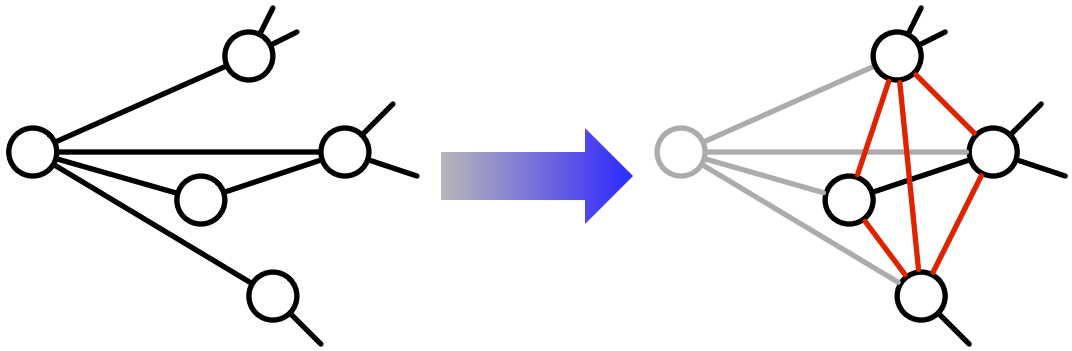
Remember that adding edges always removes conditional independencies and enlarges the family of distributions.

There are many ways to add chords; in general finding the best triangulation is NP-complete.

Here we describe one method of triangulation called **variable elimination**.

An undirected graph where every loop of size > 4 has at least one chord is called **chordal** or **triangulated**.

Variable Elimination of Undirected Graphs

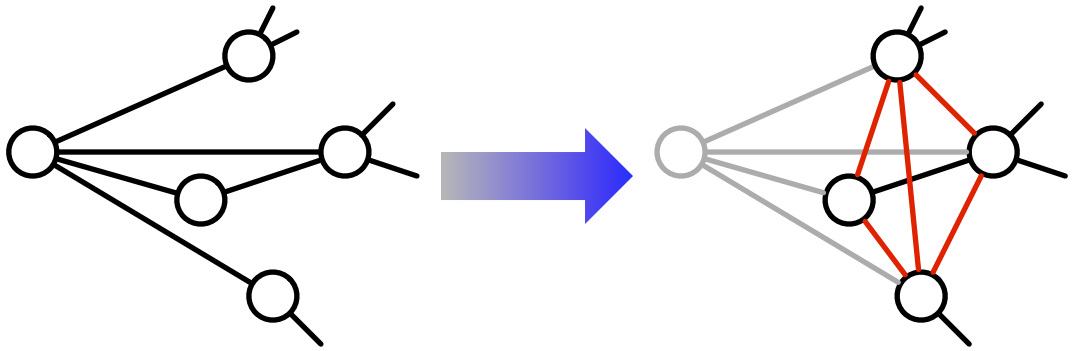


Say we compute the marginal distribution of a variable by brute force—sum out all other variables one by one (eliminate it from the graph).
Let the order of elimination be $Y_{\sigma(1)}, Y_{\sigma(2)}, \ldots, Y_{\sigma(n)}$ with $Y_{\sigma(n)}$ being the variable whose marginal distribution we are interested in.

$$
\begin{aligned}
p(Y_{\sigma(n)}) = \sum_{Y_{\sigma(n-1)}} \cdots \sum_{Y_{\sigma(1)}} p(\mathbf{Y}) &= \frac{1}{Z} \sum_{Y_{\sigma(n-1)}} \cdots \sum_{Y_{\sigma(2)}} \sum_{Y_{\sigma(1)}} \prod_{i} f_i(Y_{C_i}) \\
&= \frac{1}{Z} \sum_{Y_{\sigma(n-1)}} \cdots \sum_{Y_{\sigma(2)}} \prod_{i:C_i \not\ni \sigma(1)} f_i(Y_{C_i}) \sum_{Y_{\sigma(1)}} \prod_{i:C_i \ni \sigma(1)} f_i(Y_{C_i}) \\
&= \frac{1}{Z} \sum_{Y_{\sigma(n-1)}} \cdots \sum_{Y_{\sigma(2)}} \prod_{i:C_i \not\ni \sigma(1)} f_i(Y_{C_i}) f_{\text{new}}(Y_{C_{\text{new}}})
\end{aligned}
$$

where $C_{\text{new}} = \text{ne}(i)$, and the edges are added to the graph connecting all nodes in C_{new}.

Variable Elimination of Undirected Graphs



After we have eliminated all variables, go back to original graph, and add in all edges added during elimination.

Theorem: the graph with elimination edges added in is chordal.
Proof: by induction on the number of nodes in the graph.

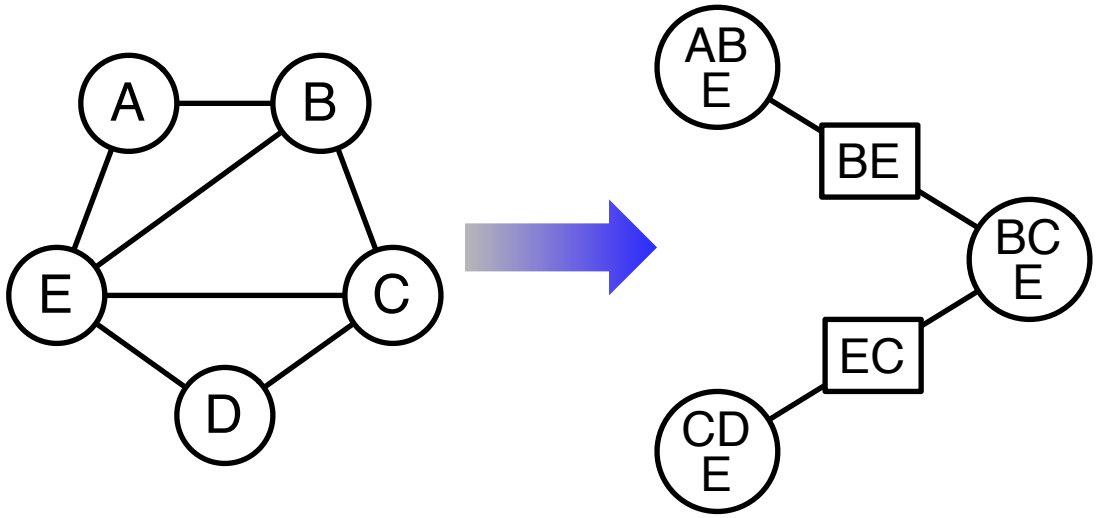
Finding a good triangulation is related to finding a good elimination ordering $\sigma(1), \ldots, \sigma(n)$. This is NP-complete.

Heuristics for good elimination ordering exist. Pick next variable to eliminate by:

- Minimum deficiency search: choose variable with least number of edges added.
- Maximum cardinality search: choose variable with maximum number of neighbours.

Minimum deficiency search has been empirically found to be better.

Chordal Graphs to Junction Trees



A **junction tree** (or **join tree**) is a tree where nodes and edges are labelled with sets of variables.

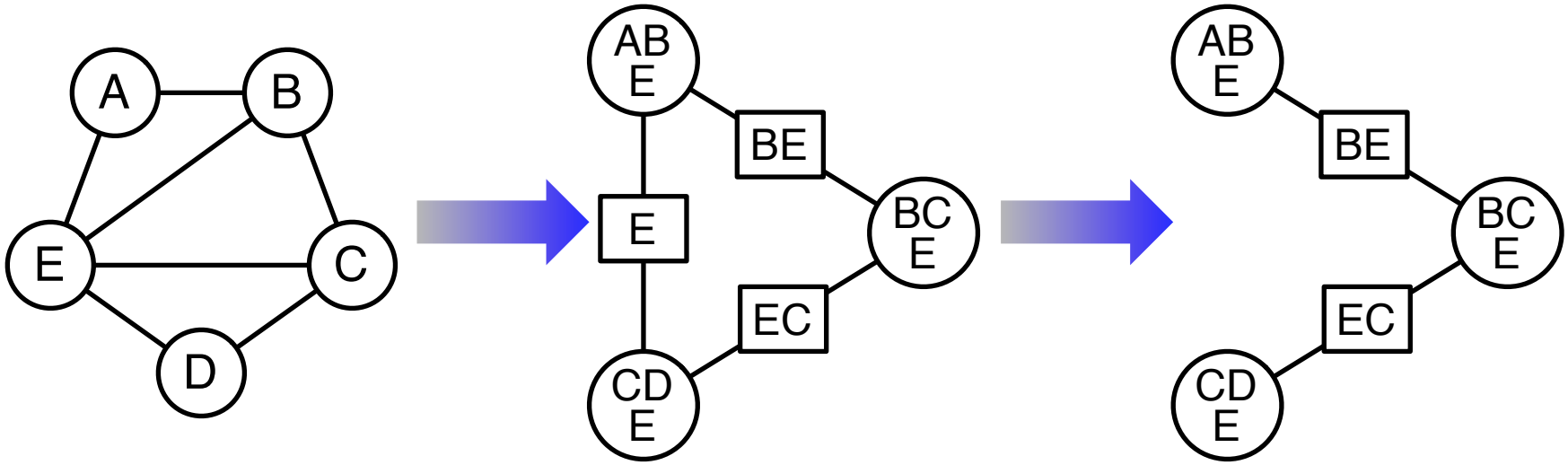
Variable sets on nodes are called cliques, and variable sets on edges are separators.

A junction tree has two properties:

- Cliques contain all adjacent separators.
- **Running intersection property**: if two cliques contain variable Y, all cliques and separators on the path between the two cliques contain Y.

The running intersection property is required for consistency.

Chordal Graphs to Junction Trees



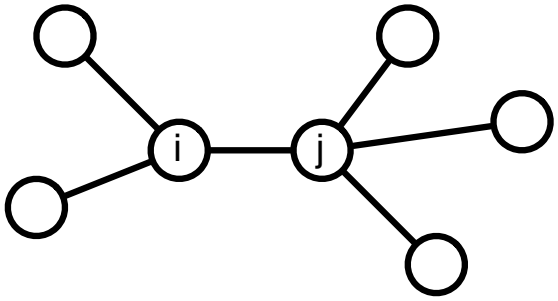
The following procedure converts a chordal graph to a junction tree:

1. Find the set of maximal cliques $C_1, \ldots, C_k$ (each of these cliques consists of an eliminated variable and its neighbours, so finding maximal cliques is easy).
2. Construct a weighted graph, with nodes labelled by the maximal cliques, and edges labelled by intersection of adjacent cliques.
3. Define the weight of an edge to be the size of the separator.
4. Run maximum weight spanning tree on the weighted graph.
5. The maximum weight spanning tree will be the junction tree.

Recap: Belief Propagation on Undirected Trees

Undirected tree parameterization for joint distribution:

$$p(\mathbf{Y}) = \frac{1}{Z} \prod_{\text{edges } (ij)} f_{(ij)}(Y_i, Y_j)$$



Define $T_{i \to j}$ to be the subtree containing i if j is removed. Define **messages**:

$$M_{i \to j}(Y_j) = \sum_{Y_{T_{i \to j}}} f_{(ij)}(Y_i, Y_j) \prod_{\text{edges } (i'j') \text{ in } T_{i \to j}} f_{(i'j')}(Y_{i'}, Y_{j'})$$

Then the messages can be recursively computed as follows:

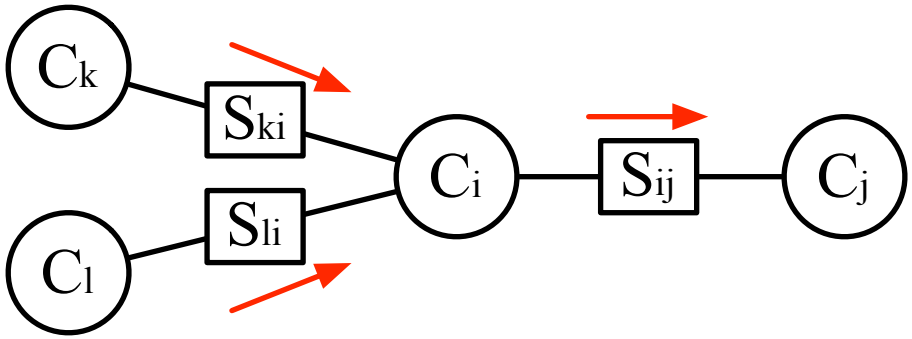
$$M_{i \to j}(Y_j) = \sum_{Y_i} f_{(ij)}(Y_i, Y_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \to i}(Y_i)$$

and:

$$p(Y_i) \propto \prod_{k \in \text{ne}(i)} M_{k \to i}(Y_i)$$

$$p(Y_i, Y_j) \propto f_{(ij)}(Y_i, Y_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \to i}(Y_i) \prod_{l \in \text{ne}(j) \setminus i} M_{l \to j}(Y_j)$$

Message Passing on Junction Trees



Since maximal cliques in the chordal graph are nodes of the junction tree, we have:

$$p(\mathbf{Y}) = \frac{1}{Z} \prod_i f_i(Y_{C_i})$$

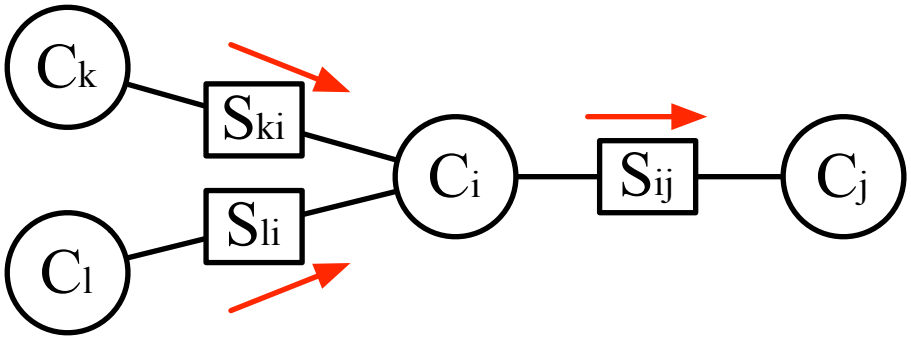
where C_i ranges over the cliques of the junction tree.

Let $S_{ij} = C_i \cap C_j$ be the separator between cliques i and j.
Let $T_{i \to j}$ be the union of cliques on the i side of j.

Define **messages**:

$$M_{i \to j}(Y_{S_{ij}}) = \sum_{Y_{T_{i \to j} \setminus C_j}} \prod_{k : C_k \subset T_{i \to j}} f_k(Y_{C_k})$$

Message Passing on Junction Trees



Messages can be computed recursively by:

$$M_{i\to j}(Y_{S_{ij}}) = \sum_{Y_{C_i \setminus S_{ij}}} f_i(Y_{C_i}) \prod_{k \in \mathsf{ne}(i)\setminus j} M_{k\to i}(Y_{S_{ki}})$$

And marginal distributions on cliques and separators are:

$$p(Y_{C_i}) = f_i(Y_{C_i}) \prod_{k \in \mathsf{ne}(i)} M_{k\to i}(Y_{S_{ki}})$$

$$p(Y_{S_{ij}}) = M_{i\to j}(Y_{S_{ij}}) M_{j\to i}(Y_{S_{ij}})$$

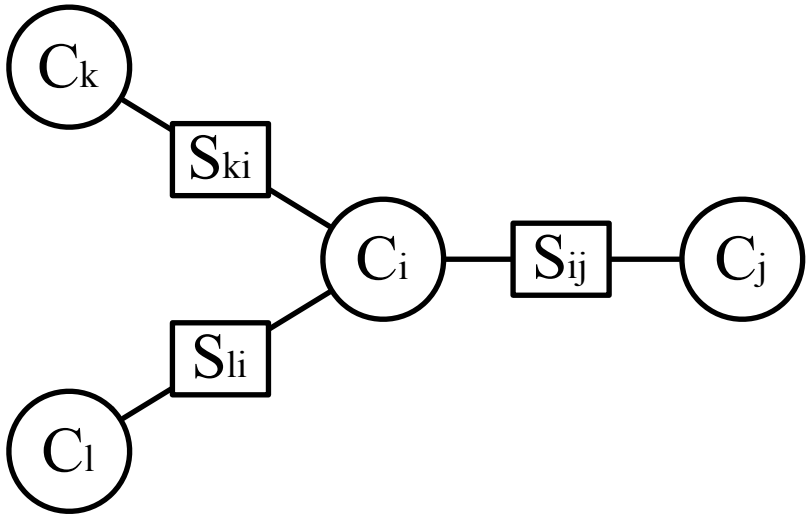
This is called **Shafer-Shenoy propagation**.

Consistency and Parameterization on Junction Trees

Because of the running intersection property and because junction trees are trees, **local consistency** of marginal distributions between cliques and separators guarantees **global consistency**.

If $q_i(Y_{C_i})$, $r_{ij}(Y_{S_{ij}})$ are distributions such that

$$\sum_{Y_{C_i} \setminus S_{ij}} q_i(Y_{C_i}) = r_{ij}(Y_{S_{ij}})$$



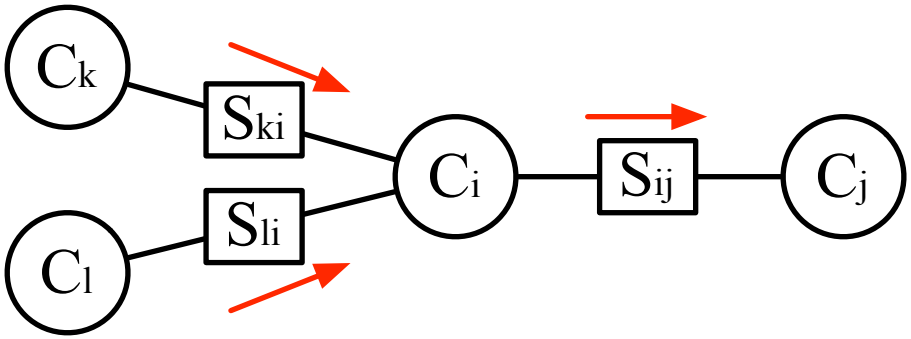
Then the following

$$p(\mathbf{Y}) = \frac{\prod_{\text{cliques } i} q_i(Y_{C_i})}{\prod_{\text{separators } (ij)} r_{ij}(Y_{S_{ij}})}$$

is also a distribution (non-negative and sums to one) such that:

$$q_i(Y_{C_i}) = \sum_{\mathbf{Y} \setminus Y_{C_i}} p(\mathbf{Y}) \qquad r_{ij}(Y_{S_{ij}}) = \sum_{\mathbf{Y} \setminus Y_{S_{ij}}} p(\mathbf{Y})$$

Reparameterization on Junction Trees



Hugin propagation is a different (but equivalent) message passing algorithm. It is based upon the idea of **reparameterization**. Initialize:

$$q_i(Y_{C_i}) \propto f_i(Y_{C_i}) \qquad\qquad r_{ij}(Y_{S_{ij}}) \propto 1$$

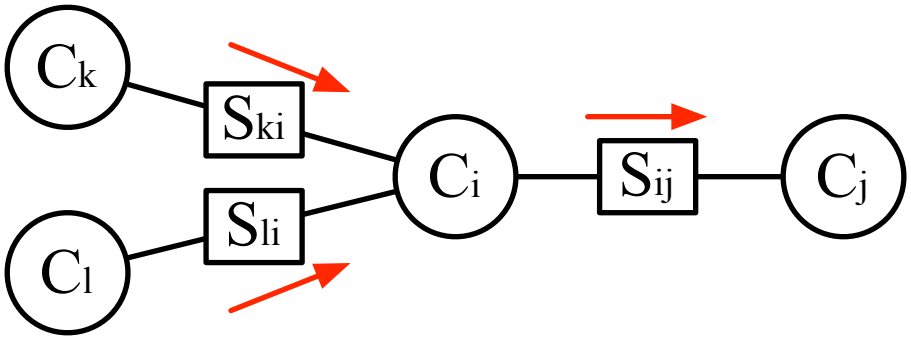
Then our probability distribution is initially

$$p(\mathbf{Y}) \propto \frac{\prod_{\text{cliques } i} q_i(Y_{C_i})}{\prod_{\text{separators } (ij)} r_{ij}(Y_{S_{ij}})}$$

A Hugin propagation update for $i \to j$ is:

$$r_{ij}^{\text{new}}(Y_{S_{ij}}) = \sum_{Y_{C_i \setminus S_{ij}}} q_i(Y_{C_i}) \qquad\qquad q_j^{\text{new}}(Y_{C_j}) = q_j(Y_{C_j}) \frac{r_{ij}^{\text{new}}(Y_{S_{ij}})}{r_{ij}(Y_{S_{ij}})}$$

Reparameterization on Junction Trees



Some properties of Hugin propagation:

- The defined distribution $p(\mathbf{Y})$ is unchanged by the updates.
- Each update introduces a local consistency constaint:

$$\sum_{Y_{C_j \setminus S_{ij}}} q_j(Y_{C_j}) = r_{ij}(Y_{S_{ij}})$$

- If each update $i \to j$ is carried out only after incoming updates $k \to i$ have been carried out, then each update needs only be carried out once.
- Each Hugin update is equivalent to the corresponding Shafer-Shenoy update.

Computational Costs of the Junction Tree Algorithm

Most of the computational cost of the junction tree algorithm is incurred during the message passing phase.

The running and memory costs of the message passing phase is $O(\sum_i |\mathcal{Y}_{C_i}|)$. This can be significantly (exponentially) more efficient than brute force.

The variable elimination ordering heuristic can have very significant impact on the message passing costs.

For certain classes of graphical models (e.g. 2D lattice Markov random field) it is possible to hand-craft an efficient ordering.

Tomorrow: approximate methods for inference, for when junction tree is impractical.

End Notes

Textbooks on graphical models:

Koller and Friedman (2009):
Probabilistic Graphical Models: Principles and Techniques. MIT Press.

Cowell, Dawid, Lauritzen and Spiegelhalter (2007). Probabilistic Networks and Expert Systems. Springer.

Jensen and Graven-Nielsen (2007). Bayesian Networks and Decision Graphs. Springer.

End Notes