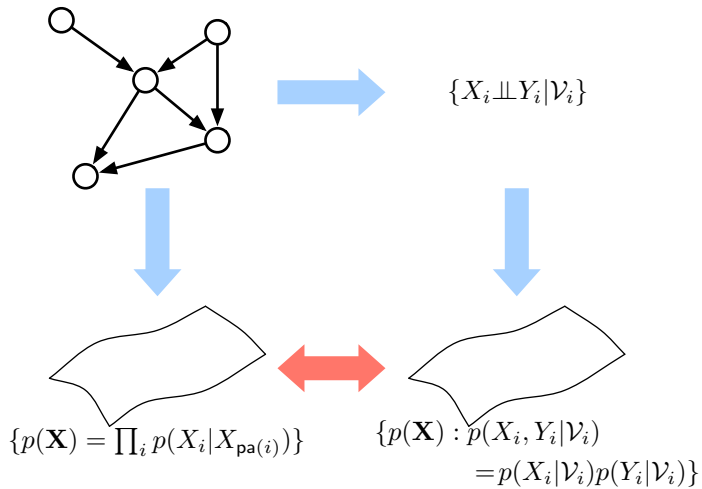


## Recap on Graphical Models



## Inference in Graphical Models

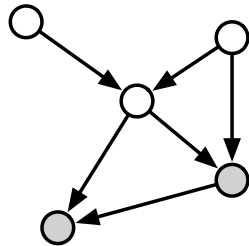
Thus far, we have used graphical models to encode the conditional independencies and parameterizations of probability distributions visually. Can they do more?

We often need to compute a function of the distribution on hidden nodes conditioned on some observed ones.

- marginals:  $p(A|DE), \dots$
- most likely values:  $\text{argmax} p(ABC|DE)$

Message passing algorithms exploit conditional independence relationships to make this computation efficient. Examples:

- forward-backward on HMMs and SSMs,
- Viterbi on HMMs and SSMS,
- Belief Propagation on undirected trees.



Today we will learn about message-passing algorithms that can work on arbitrary graphs. Specifically we will try to compute marginal distributions over single variables.

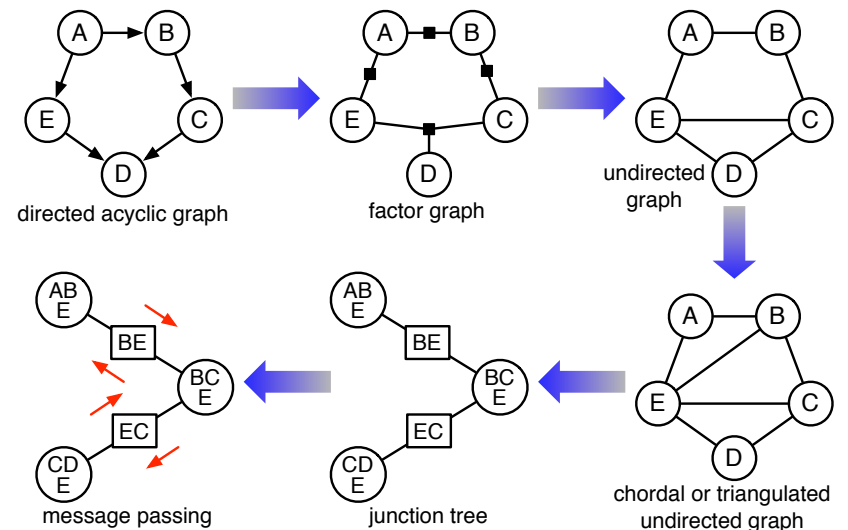
## Graph Transformations

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**).

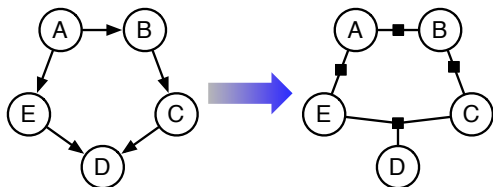
Suppose we are interested in inference in a distribution  $p(\mathbf{x})$  representable in a graphical model  $G$ .

- In transforming  $G$  to an easy-to-handle  $G'$ , we need to ensure that  $p(\mathbf{x})$  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  $p(\mathbf{x})$  will be in the family of distributions represented by  $G'$ .
- Thus inference algorithms working on  $G'$  will work for  $p(\mathbf{x})$  too.

## The Junction Tree Algorithm



### Directed Acyclic Graphs to Factor Graphs



Factors are simply the conditional distributions in the DAG.

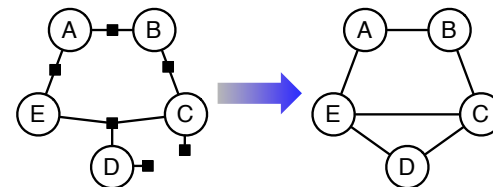
$$p(\mathbf{X}) = \prod_i p(X_i | X_{\text{pa}(i)})$$

$$= \prod_i f_i(X_{C_i})$$

where  $C_i = i \cup \text{pa}(i)$  and  $f_i(X_{C_i}) = p(X_i | X_{\text{pa}(i)})$ .

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

### Factor Graphs to Undirected Graphs



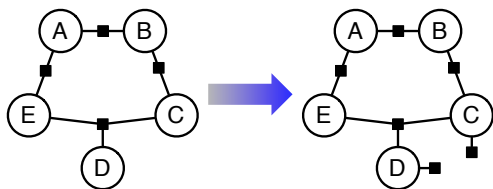
Just need to make sure that every factor is contained in some maximal clique of the undirected graph.

$$p(\mathbf{X}) = \frac{1}{Z} \prod_i f_i(X_{C_i})$$

We can make sure of this simply by converting each factor into a clique, and absorbing  $f_i(X_{C_i})$  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.

### Entering Evidence in Factor Graphs



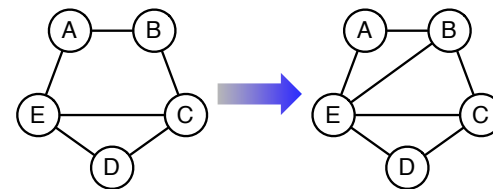
Often times we are interested in inferring posterior distributions given observed evidence (e.g.  $D = \text{wet}$  and  $C = \text{rain}$ ).

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

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

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

### 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.

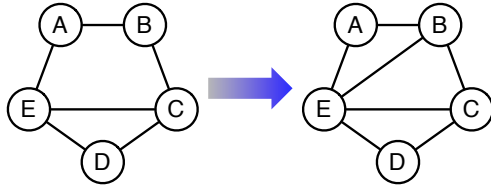
If loops (cycles) are present in the graph, message passing can lead to overconfidence due to double counting of information, and to oscillatory (non-convergent) behaviour<sup>2</sup>.

To prevent this overconfident and oscillatory behaviour, we need to make sure that different channels of information **communicate** 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.

<sup>2</sup>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.

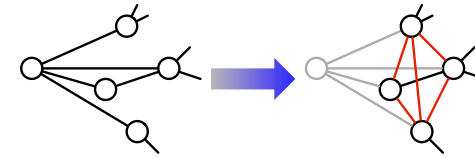
Remember that adding edges always removes conditional independencies and enlarges the family of distributions.

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



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.

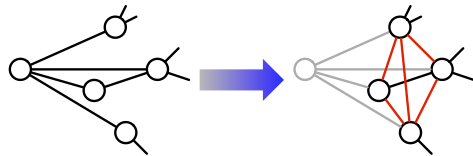
Finding a good triangulation is related to finding a good elimination ordering  $\sigma(1), \dots, \sigma(n)$ . This is NP-complete.

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.

## Variable Elimination of Undirected Graphs



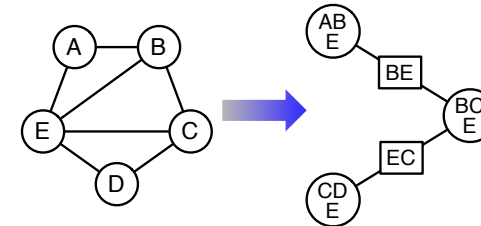
Say we compute the marginal distribution of a variable by brute force—sum out all other variables one by one (eliminate it from the graph).

Let the order of elimination by  $X_{\sigma(1)}, X_{\sigma(2)}, \dots, X_{\sigma(n)}$  with  $X_{\sigma(n)}$  being the variable whose marginal distribution we are interested in.

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

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

## Chordal Graphs to Junction Trees



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

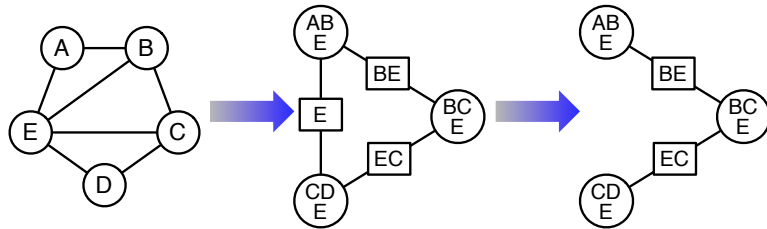
Variable sets on nodes are called **cliques**, and Variable sets on edges are **separators**.

A junction tree has two properties:

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

The running intersection property is required for consistency.

## Chordal Graphs to Junction Trees



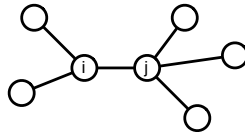
The following procedure converts a chordal graph to a junction tree:

1. Find the set of maximal cliques  $C_1, \dots, C_k$  (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:

$$p(\mathbf{X}) = \frac{1}{Z} \prod_{\text{edges } (ij)} f_{(ij)}(X_i, X_j)$$



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

$$M_{i \rightarrow j}(X_j) = \sum_{X_i} f_{(ij)}(X_i, X_j) \prod_{\text{edges } (i'j') \text{ in } T_{i \rightarrow j}} f_{(i'j')}(X_{i'}, X_{j'})$$

Then the messages can be recursively computed as follows:

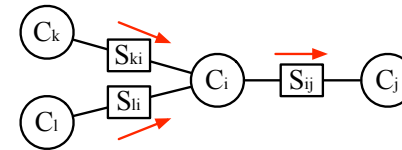
$$M_{i \rightarrow j}(X_j) = \sum_{X_i} f_{(ij)}(X_i, X_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \rightarrow i}(X_i)$$

and:

$$p(X_i) \propto \prod_{k \in \text{ne}(i)} M_{k \rightarrow i}(X_i)$$

$$p(X_i, X_j) \propto f_{(ij)}(X_i, X_j) \prod_{k \in \text{ne}(i) \setminus j} M_{k \rightarrow i}(X_i) \prod_{l \in \text{ne}(j) \setminus i} M_{l \rightarrow j}(X_j)$$

## Message Passing on Junction Trees



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

$$p(\mathbf{X}) = \frac{1}{Z} \prod_i f_i(X_{C_i})$$

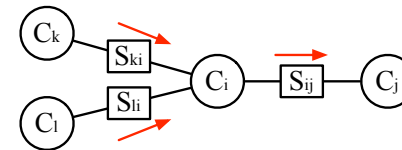
where  $C_i$  ranges over the cliques of the junction tree.

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

Define **messages**:

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

## Message Passing on Junction Trees



Messages can be computed recursively by:

$$M_{i \rightarrow j}(X_{S_{ij}}) = \sum_{X_{C_i \setminus S_{ij}}} f_i(X_{C_i}) \prod_{k \in \text{ne}(i) \setminus j} M_{k \rightarrow i}(X_{S_{ki}})$$

And marginal distributions on cliques and separators are:

$$p(X_{C_i}) = f_i(X_{C_i}) \prod_{k \in \text{ne}(i)} M_{k \rightarrow i}(X_{S_{ki}})$$

$$p(X_{S_{ij}}) = M_{i \rightarrow j}(X_{S_{ij}}) M_{j \rightarrow i}(X_{S_{ij}})$$

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

## Consistency and Parameterization on Junction Trees

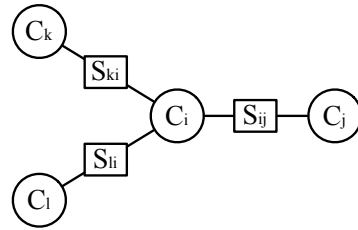
Because of the running intersection property and because junction trees are trees, **local consistency** of marginal distributions between cliques and separators guarantees **global consistency**.

If  $q(X_{C_i}), r(X_{S_{ij}})$  are distributions such that

$$\sum_{X_{C_i \setminus S_{ij}}} q(X_{C_i}) = r(X_{S_{ij}})$$

Then the following

$$p(\mathbf{X}) = \frac{\prod_{\text{cliques } i} q(X_{C_i})}{\prod_{\text{separators } (ij)} r(X_{S_{ij}})}$$

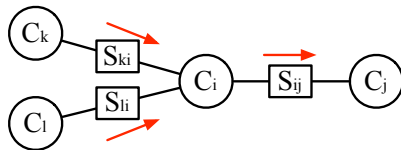


is also a distribution (non-negative and sums to one) such that:

$$q(X_{C_i}) = \sum_{\mathbf{X} \setminus X_{C_i}} p(\mathbf{X})$$

$$r(X_{S_{ij}}) = \sum_{\mathbf{X} \setminus X_{S_{ij}}} p(\mathbf{X})$$

## Reparameterization on Junction Trees



**Hugin propagation** is a different (but equivalent) message passing algorithm.

It is based upon the idea of **reparameterization**. Initialize:

$$q(X_{C_i}) \propto f_i(X_{C_i}) \quad r(X_{S_{ij}}) \propto 1$$

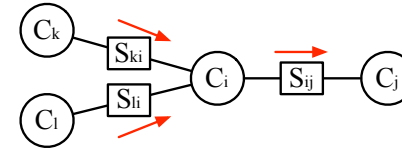
Then our probability distribution is initially

$$p(\mathbf{X}) \propto \frac{\prod_{\text{cliques } i} q(X_{C_i})}{\prod_{\text{separators } (ij)} r(X_{S_{ij}})}$$

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

$$r^{\text{new}}(X_{S_{ij}}) = \sum_{X_{C_i \setminus S_{ij}}} q(X_{C_i}) \quad q^{\text{new}}(X_{C_j}) = q(X_{C_j}) \frac{r^{\text{new}}(X_{S_{ij}})}{r(X_{S_{ij}})}$$

## Reparameterization on Junction Trees



Some properties of Hugin propagation:

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

$$\sum_{X_{C_i \setminus S_{ij}}} q(X_{C_i}) = r(X_{S_{ij}})$$

- If each update  $i \rightarrow j$  is carried out only after incoming updates  $k \rightarrow i$  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  $O(\sum_i |\mathcal{X}_{C_i}|)$ . 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.

## Other Inference Algorithms

There are other approaches to inference in graphical models which may be more efficient under specific conditions:

**Cutset conditioning:** or “reasoning by assumptions”. Find a small set of variables which, if they were given (i.e. known) would render the remaining graph “simpler”. For each value of these variables run some inference algorithm on the simpler graph, and average the resulting beliefs with the appropriate weights.

**Loopy belief propagation:** just use belief propagation eventhough there are loops. No guarantee of convergence, but often works well in practice. Some (weak) guarantees about the nature of the answer if the message passing *does* converge.

Second half of course: we will learn about a variety of **approximate inference** algorithms when the graphical model is so large/complex that no exact inference algorithm can work efficiently.

## Learning in Graphical Models

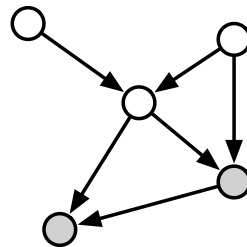
In combination with an appropriate message passing algorithm, the factored structure implied by the graph also makes learning easy.

Consider data points comprising observations of a subset of variables.  
ML learning  $\Rightarrow$  adjust parameters to maximise:

$$\begin{aligned}\mathcal{L} &= p(X_{\text{obs}}|\theta) \\ &= \sum_{X_{\text{unobs}}} p(X_{\text{obs}}, X_{\text{unobs}}|\theta)\end{aligned}$$

by EM, need to maximise

$$\begin{aligned}\mathcal{F} &= \left\langle \log p(X_{\text{obs}}, X_{\text{unobs}}|\theta) \right\rangle_{p(X_{\text{unobs}}|X_{\text{obs}})} \\ &= \left\langle \sum_i \log f_i(X_{C_i}|\theta_i) - \log Z \right\rangle_{p(X_{\text{unobs}}|X_{\text{obs}})} \\ &= \sum_i \left\langle \log f_i(X_{C_i}|\theta_i) \right\rangle_{p(X_{C_i} \setminus X_{\text{obs}}|X_{\text{obs}})} - \log Z\end{aligned}$$



So learning only requires posterior marginals on cliques (obtained by messaging passing) and updates on cliques; c.f. the Baum-Welch procedure for HMMs.