Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.438 Algorithms For Inference
Fall 2014

**Suggested Reading:**

- "Probabilistic Modelling and Reasoning: The Junction Tree Algorithm" (Barber, 2004). (Ihler, Fisher, Willsky, 2005).

# Recitation 8: Junction Trees

## Contents

# 1 The Junction Tree Algorithm

For general undirected graphs potentially with loops, we sought an exact inference algorithm for computing all the marginals. Simply running the elimination algorithm to obtain the marginal at each node works but does not provide a clear way of recycling intermediate computations. The junction tree algorithm is a principled way to compute marginals for every node in any undirected graphical model and makes it clear how intermediate computations are stored.

Consider graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $|\mathcal{V}| = N$. We will assume that we are given a set of potentials $\phi_C(x_C)$ specified directly over the maximal cliques of the graph. Thus, we implicitly assume that we have a list of the maximal cliques given to us.

The junction tree algorithm is:

1. **Chordalize the graph using the elimination algorithm with an arbitrary elimination ordering, if required.** We first check whether the graph is chordal. This can be done either by inspection, or more systematically, by trying to find an elimination ordering which does not add any edges. If the original graph is chordal, it is always possible to find a good elimination order (i.e. an order where no edges are added) simply by eliminating those nodes whose neighbors are fully connected. On the other hand, if the graph is not chordal, we will find that it is impossible to eliminate all the nodes without adding edges. In this case, we carry out elimination using 'some' elimination ordering, and add all these edges to our original graph. In other words, we work with the reconstituted graph instead of the original, since the latter is always chordal.

   Running time: $O(N^2)$.

2. **Find the maximal cliques in the chordal graph.** If the original graph was chordal, this step does not need to be carried out, since the maximal cliques are already given to us (by assumption). However, if the triangulated graph we use to build our junction tree differs from the original, we need to find the maximal cliques of this new chordal graph. In most cases, these can be found simply by inspection. However, there is also an extremely simple and efficient algorithm to find maximal cliques for chordal graphs, which we shall discuss in a later section.
   *Note:* For a chordal graph, the total number of maximal cliques in the graph is $\leq N$.

   Let $\mathcal{C}$ be the set of maximal cliques obtained by the above procedure, and we will denote $D = \max_{C \in \mathcal{C}} |C|$ to be the size of the largest maximal clique in $\mathcal{C}$.

   Running time: $O(N^2)$.

3. **Compute the separator sets for each pair of maximal cliques and construct a weighted clique graph.** For each pair of maximal cliques $(C_i, C_j)$ in the graph, we check whether they possess any common variables. If yes, we designate a separator set between these 2 cliques as $S_{ij} = C_i \cap C_j$.

While computing these separator sets, we simultaneously build a clique graph (not necessarily a tree!) by adding edge $(C_i, C_j)$ with weight $|C_i \cap C_j|$ if this weight is positive.

This step can be implemented quickly in practice using a hash table.

Running time: $O(|\mathcal{C}|^2 D) = O(N^2 D)$.

4. **Compute a maximum-weight spanning tree on the weighted clique graph to obtain a junction tree.** Finding maximum weight spanning trees is a well studied problem, for which the two (greedy) algorithms of choice are Kruskal's algorithm and Prim's algorithm. [1] We will give Kruskal's algorithm for finding the maximum-weight spanning tree:

   - Initialize an edgeless graph $\mathcal{T}$ with nodes that are all the maximal cliques in our chordal graph. We'll basically add edges to $\mathcal{T}$ until it becomes a junction tree.
   - Sort the $m$ edges in our clique graph from step 3 by weight so that if edge $e_i$ has weight $w_j$, then we have $e_1, e_2, \ldots, e_m$ with $w_1 \geq w_2 \geq \cdots \geq w_m$.
   - For $i = 1, 2, \ldots, m$:
     - Add edge $e_i$ to $\mathcal{T}$ if it does not introduce a cycle.[2]
     - If $|\mathcal{C}| - 1$ edges have been added, quit.

     The resulting tree from the above procedure is a junction tree. The time complexity of Kruskal's algorithm is equal to $O(|E| \log |E|)$ operations where $|E|$ is the number of edges in our clique graph from step 3. This complexity is essentially due to the sorting step. Note that $|E| = O(|\mathcal{C}|^2)$. Thus, $O(|E| \log |E|) = O(|\mathcal{C}|^2 \log |\mathcal{C}|^2) = O(2|\mathcal{C}|^2 \log |\mathcal{C}|) = O(|\mathcal{C}|^2 \log |\mathcal{C}|)$.

   Running time: $O(|\mathcal{C}|^2 \log |\mathcal{C}|) = O(|N|^2 \log |N|)$.

5. **Define potentials on the junction tree.** If the original distribution was $p_{\mathbf{x}}(\mathbf{x}) \propto \prod_{C \in \mathcal{C}} \phi_C(x_C)$, then the junction tree $\mathcal{T} = (\mathcal{C}, \mathcal{F})$ has factorization

$$p_{\mathbf{y}}(y_{C_1}, y_{C_2}, \ldots, y_{C_{|\mathcal{C}|}}) \propto \prod_{C \in \mathcal{C}} \underbrace{\phi_C(y_C)}_{\text{singleton potential}} \prod_{(C_i, C_j) \in \mathcal{F}} \underbrace{\mathbf{1}\{y_{C_i} \text{ and } y_{C_j} \text{ agree on indices corresponding to } C_i \cap C_j\}}_{\text{edge potential } \psi_{C_i, C_j}}.$$

[1] Often greedy algorithms can be shown to fail miserably on specific inputs, but this is not the case for Kruskal's and Prim's algorithms, which are celebrated examples of greedy algorithms that are in fact optimal for the problems they are trying to solve. Note that maximum spanning trees may not be unique.

[2] As an aside, with an input graph over $|V|$ nodes, checking that adding an edge does not introduce a cycle can be done in $O(\log^* |V|)$ time, where $\log^*$ (pronounced "log star" and also referred to as the iterated logarithm) is the number of times log needs to be applied to a number to get a value less than or equal to 1 (see Chapter 5 of S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani's "Algorithms". The growth of $\log^*$ is ridiculously slow and for all intents and purposes, $\log^*$ might as well be a constant. For example, $\log^*(2) = 1$ and $\log^*(2^{65536}) = 5$, where we are assuming log to be base 2.

*Remark 1*: This does not actually have to be implemented; if we keep track of separator sets, while running sum-product, we can just enforce agreement and avoid having to do extra summations. More details for this are in the next step.

6. **Run sum-product on the junction tree. (Shafer-Shenoy algorithm)** The largest alphabet size at a node is $|\mathcal{X}|^D$, so the running time is $O(|\mathcal{C}||\mathcal{X}|^D)$. To account for the remark made in the previous step, suppose $V$ and $W$ are two neighboring maximal-clique nodes in the junction tree with separator set $S = V \cap W$. Let $x_V = (x_S, x_{V\setminus S})$ be a value maximal clique $\mathbf{x}_V$ can take on, and let $y_W = (y_S, y_{W\setminus S})$ be a value maximal clique $\mathbf{x}_W$ can take on. Then the message sent from $V$ to $W$ is:

$$\begin{aligned}
m_{V\to W}(y_S, y_{W\setminus S}) &\equiv m_{V\to W}(y_W) \\
&= \sum_{x_V} \phi_V(x_V)\psi_{V,W}(x_V, y_W) \prod_{U\in N(V)\setminus W} m_{U\to V}(x_V) \\
&= \sum_{x_V} \phi_V(x_S, x_{V\setminus S})\psi_{V,W}(x_S, x_{V\setminus S}, y_S, y_{W\setminus S}) \prod_{U\in N(V)\setminus W} m_{U\to V}(x_S, x_{V\setminus S}) \\
&= \sum_{x_V} \phi_V(x_S, x_{V\setminus S})\mathbf{1}\{x_S = y_S\} \prod_{U\in N(V)\setminus W} m_{U\to V}(x_S, x_{V\setminus S}) \\
&= \sum_{x_{V\setminus S}} \phi_V(y_S, x_{V\setminus S}) \prod_{U\in N(V)\setminus W} m_{U\to V}(y_S, x_{V\setminus S}).
\end{aligned}$$

By noticing that we can just write the message in this way, we need not explicitly store edge potentials!

*Remark 2:* This particular style of modified message passing on junction trees is known as the Shafer-Shenoy algorithm.

Running time: $O(|\mathcal{C}||\mathcal{X}|^D) = O(N|\mathcal{X}|^D)$.

7. **Compute marginals.** After running sum-product on the junction tree, we have marginals for maximal cliques, but to actually get the marginal for each node in our original graph, we need to essentially use brute-force to obtain node marginals given a maximal-clique marginal. For example, say we obtain marginal $p_{\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3}(x_1, x_2, x_3)$ over maximal clique $\{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3\}$. Then to compute the marginal for $\mathsf{x}_1$, we just compute $p_{\mathsf{x}_1}(x_1) = \sum_{x_2,x_3} p_{\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3}(x_1, x_2, x_3)$. So for each node $v$ in our original graph, we need to do $O(|\mathcal{X}|^{|C|})$ operations to compute the marginal for $v$, where $C$ is the smallest maximal clique that $v$ participates in. Thus, the maximum possible running time for doing this across all nodes is: $O(N|\mathcal{X}|^{\max_i |C_i|}) = O(N|\mathcal{X}|^D)$.
Running time: $O(N|\mathcal{X}|^D)$.

So the overall running time for the whole algorithm is: $O(N^2D + N^2 \log N + N|\mathcal{X}|^D)$.

Note that the largest maximal clique size $D$ of the chordal graph depends on the elimination ordering, where the smallest $D$ possible across all elimination orderings is called the tree-width of graph $\mathcal{G}$. So the Junction Tree Algorithm takes time exponential in $D$

for exact inference. But this isn't a surprising result since we can encode NP-complete problems as inference problems over undirected graphs (e.g., 3-coloring), and the P vs. NP problem would have long ago been settled if the Junction Tree Algorithm efficiently solved all inference problems.

## 2 Hugin updates (optional, not discussed in recitation)

We next look at a different way to obtain marginal distributions over maximal cliques in the chordal graph. The lecture coverage for the junction tree algorithm intentionally aimed for an intuitive exposition, leading to the above algorithm. However, the junction tree algorithm more generally describes several algorithms that do essentially the same thing but vary in implementation details.

The algorithm described above where we have simplified the message-passing update so that we don't have to store edge potentials is the Shafer-Shenoy algorithm ($\sim$ 1990). Another algorithm is the Hugin algorithm ( $\sim$ 1990), which does steps 5 and 6 above differently; the trick used by the Hugin algorithm is the same as the trick we used for parallel Sum-Product to reduce the number of message multiplications. However, with some clever bookkeeping, the update equations for the Hugin algorithm can look a tad bit foreign compared to what we've seen thus far in the class because these update equations actually sequentially modify node and edge potentials!

The basic idea is as follows. Recall that the Shafer-Shenoy message update equation is

$$m_{V\to W}^{t+1}(y_S, y_{W\setminus S}) = \sum_{x_{V\setminus S}} \phi_V(y_S, x_{V\setminus S}) \prod_{U\in N(V)\setminus W} m_{U\to V}^{t}(y_S, x_{V\setminus S}), \tag{1}$$

where we now explicitly write out the iteration number to make it clear what's being updated in the following derivation. Define

$$\widetilde{\phi}_V^t(x_S, x_{V\setminus S}) = \phi_V(x_S, x_{V\setminus S}) \prod_{U\in N(V)} m_{U\to V}^{t}(x_S, x_{V\setminus S}). \tag{2}$$

Combining Eqs. (1) and (2) gives

$$m_{V\to W}^{t+1}(y_S, y_{W\setminus S}) = \sum_{x_{V\setminus S}} \frac{\widetilde{\phi}_V^t(y_S, x_{V\setminus S})}{m_{W\to V}^{t}(y_S, x_{V\setminus S})}.$$

Note that this message update does not actually depend on value $y_{W\setminus S}$, so this message behaves like a potential over separator set $S$. Once we compute the above update, we can update $\widetilde{\phi}_V^t(x_S, x_{V\setminus S})$ to account for the new message $m_{V\to W}^{t+1}(y_S, y_{W\setminus S})$:

$$\begin{aligned}
\widetilde{\phi}_W^{t+1}(y_S, x_{W\setminus S}) &= \phi_W(y_S, x_{W\setminus S}) \left( \prod_{U\in N(W)\setminus V} m_{U\to W}^{t}(y_S, x_{W\setminus S}) \right) m_{V\to W}^{t+1}(y_S, y_{W\setminus S}) \\
&= \phi_W(y_S, x_{W\setminus S}) \left( \prod_{U\in N(W)} m_{U\to W}^{t}(y_S, x_{W\setminus S}) \right) \frac{m_{V\to W}^{t+1}(y_S, y_{W\setminus S})}{m_{V\to W}^{t}(y_S, y_{W\setminus S})} \\
&= \widetilde{\phi}_W^{t}(y_S, x_{W\setminus S}) \frac{m_{V\to W}^{t+1}(y_S, y_{W\setminus S})}{m_{V\to W}^{t}(y_S, y_{W\setminus S})}.
\end{aligned}$$

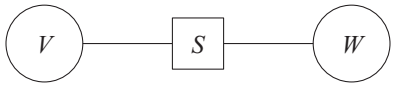
So it's possible to keep track of only the messages sent (which behave as potentials over separator sets) and modified node potentials $\widetilde{\phi}_V$'s for each maximal clique $V$. As a warning up front, the Hugin update equations will still look slightly different because via careful bookkeeping, it does not separately store $m_{V\to W}$ and $m_{W\to V}$ potentials.

With the above intuition, we now give the Hugin algorithm. We directly place potentials on separator sets, which are like edge potentials, and node and separator set potentials are sequentially updated until each node potential is actually proportional to the node's marginal distribution. We write our distribution as:

$$p_{\mathbf{x}}(\mathbf{x}) \propto \prod_{C\in\mathcal{C}} \phi_C(x_C) \overset{(i)}{=} \frac{\prod_{C\in\mathcal{C}} \phi_C(x_C)}{\prod_{S\in\mathcal{S}} \phi_S(x_S)} \tag{3}$$

where $\mathcal{S}$ is the set of separator sets. We initialize $\phi_S(\cdot) \equiv 1$ for all $S \in \mathcal{S}$. Sequential Hugin updates to the node and separator set potentials will ensure that equality $(i)$ above always holds true, which guarantees that at every update, the node and separator set potentials actually do describe the same joint distribution we began with.

The Hugin update is as follows, where we can view this as sending a message from maximal clique $V$ to neighboring maximal clique $W$ where we have separator set $S = V \cap W$:




$$\phi_V(x_V) = \phi_V(x_S, x_{V\setminus S}) \qquad \phi_S(x_S) \qquad \phi_W(x_W) = \phi_W(x_S, x_{W\setminus S})$$

$$\phi_S^*(x_S) = \sum_{x_{V\setminus S}} \phi_V(x_S, x_{V\setminus S})$$

$$\phi_W^*(x_W) \equiv \phi_W^*(x_S, x_{W\setminus S}) = \frac{\phi_S^*(x_S)}{\phi_S(x_S)} \phi_W(x_S, x_{W\setminus S})$$

After running these two updates, we replace $\phi_S$ with $\phi_S^*$ and $\phi_W$ with $\phi_W^*$, effectively updating the separator potential for $S$ and the node potential for $W$. We do these updates from leaves to root and then from the root back to the leaves as in the serial sum-product algorithm. At the end, we have

$$\phi_C(x_C) \propto p_{\mathbf{x}_C}(x_C) \qquad \text{for all } C \in \mathcal{C}.$$

We mention two key properties of the Hugin update equations:

1. At each update step, equality $(i)$ in Eq. (3) is preserved. To show this, it suffices to look at the terms involved in an update:

$$\frac{\phi_V(x_V)\phi_W^*(x_W)}{\phi_S^*(x_S)} = \frac{\phi_V(x_V)}{\phi_S^*(x_S)} \left( \frac{\phi_S^*(x_S)}{\phi_S(x_S)} \phi_W(x_W) \right) = \frac{\phi_V(x_V)\phi_W(x_W)}{\phi_S(x_S)},$$

   as desired.

2. There actually is consistency for two neighboring maximal cliques after messages have been sent both directions. In particular, suppose we send message from $V$ to $W$. Then we send a message from $W$ back to $V$, where using the above update equations gives:

$$\phi_S^{**}(x_S) = \sum_{W \setminus S} \phi_W^*(x_S, x_{W \setminus S})$$
$$\phi_V^*(x_V) = \frac{\phi_S^{**}(x_S)}{\phi_S^*(x_S)} \phi_V(x_V)$$

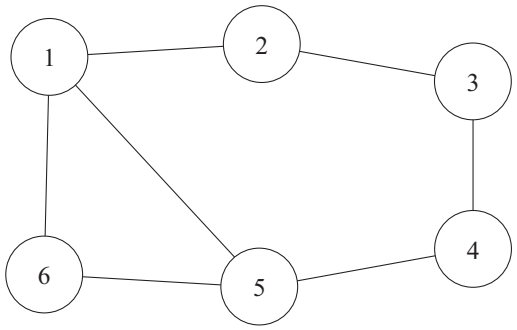
Then

$$\begin{aligned}
\sum_{V \setminus S} \phi_V^*(x_S, x_{V \setminus S}) &= \sum_{V \setminus S} \frac{\phi_S^{**}(x_S)}{\phi_S^*(x_S)} \phi_V(x_S, x_{V \setminus S}) \\
&= \sum_{V \setminus S} \frac{\sum_{W \setminus S} \phi_W^*(x_S, x_{W \setminus S})}{\phi_S^*(x_S)} \phi_V(x_S, x_{V \setminus S}) \\
&= \sum_{W \setminus S} \phi_W^*(x_S, x_{W \setminus S}) \frac{\sum_{V \setminus S} \phi_V(x_S, x_{V \setminus S})}{\phi_S^*(x_S)} \\
&= \sum_{W \setminus S} \phi_W^*(x_S, x_{W \setminus S}) \frac{\phi_S^*(x_S)}{\phi_S^*(x_S)} \\
&= \sum_{W \setminus S} \phi_W^*(x_S, x_{W \setminus S}).
\end{aligned}$$

This means that the two maximal-clique marginals for $V$ and $W$ agree on variables in their intersection $S$, as desired.

## 3 Example problem for building JCT

Check your understanding of the junction tree algorithm by running steps 1-4 (listed earlier) by hand to obtain a junction tree for the following graph:



# 4 Maximal cliques in chordal graphs

We saw in Pset-2 that there exists graphs with exponentially many maximal cliques. It is a remarkable fact that in chordal graphs, we can have no more than N maximal cliques, where N is the number of nodes in the graph. Moreover, there exists an efficient procedure to find these maximal cliques. Both these facts are closely related to the elimination algorithm. We shall state and prove both of these claims below.

**Claim 1** *A chordal graph with N vertices can have no more than N maximal cliques.*

**Claim 2** *Given a chordal graph with $G = (V, E)$, where $|V| = N$, there exists an algorithm to find all the maximal cliques of $G$ which takes no more than $O(|N|^4)$ time.*

**Definition 1 (elimination clique)** *Given a chordal graph $G$, and an elimination ordering for $G$ which does not add any edges. Suppose node $i$ is eliminated in some step of the elimination algorithm, then the clique consisting of the node $i$ along with its neighbors during the elimination step (which must be fully connected since elimination does not add edges) is called an elimination clique*[3]*.*
*More formally, suppose node $i$ is eliminated in the $k$-th step of the algorithm, and let $G_{(k)}$ be the graph just before the $k$-th elimination step. Then $C_i = \{i\} \cup \mathcal{N}_{(k)}(i)$, where $\mathcal{N}_{(k)}(i)$ represents the neighbors of $i$ in $G_{(k)}$.*

**Theorem 1** *Given a chordal graph and an elimination ordering which does not add any edges. Let $\mathcal{C}$ be the set of maximal cliques in the chordal graph, and let $\mathcal{C}_e = (\cup_{i \in V} C_i)$ be the set of elimination cliques obtained from this elimination ordering. Then, $\mathcal{C} \subseteq \mathcal{C}_e$. In other words, every maximal clique is also an elimination clique for this particular ordering.*

*Proof:*
Proof of this theorem is left as an exercise.
■

The above theorem immediately proves the 2 claims given earlier. Firstly, it shows that a chordal graph cannot have more than N maximal cliques, since we have only N elimination cliques. Further, it gives us an efficient algorithm for finding these N maximal cliques - simply go over each elimination clique and check whether it is maximal. Even if we do this in a brute force way, we will not take more than $|\mathcal{C}_e|^2 * D$ time, which is bounded by $O(N^3)$. (since both clique size and no. of elimination cliques is bounded by N). In fact, in practice we only need to check whether the clique is maximal by comparing it to cliques that appear *before* it in the elimination ordering.

The above proof may appear very simple, but we have used a property here that we did not prove: that we can find an elimination ordering for a chordal graph which does not add edges. This statement can be proved by using the following 2 observations:
i) Every chordal graph has a node whose neighbors are fully connected. Thus eliminatiing such a node will not add any edges to the graph.

[3] AFAIK, the term elimination clique is not standard, and has not been defined elsewhere in the literature. We are defining it only to simplify presentation of the topic

ii) If a graph $G$ is chordal, any graph obtained by deleting a vertex from $G$ (as well as the edges incident on that vertex), must also be chordal. In other words, deleting a node from a graph preserves chordality.

The first observation follows from Lemma-2 in lecture-14, while the second observation is easy to prove. Thus, there exists an elimination ordering for chordal graphs, which consists of always eliminating a node whose neighbors are fully connected. We can simply use a brute-force algorithm for finding the elimination ordering: For each elimination step, go through the entire set of nodes, until you find one whose neighbors are fully connected. Even this naive approach is polynomial time viz. $O(N^4)$, and it can be speeded up by using better algorithms.

*Remark 3:* The above procedure describes an $O(N^4)$ algorithm for finding the maximal cliques, but there exist much faster algorithms. I will give a reference from last year's recitation notes for an $O(N^2)$ algorithm, for those who are interested: Algorithm 2 of P. Galinier, M., Habib, and C. Paul's "Chordal graphs and their clique graphs" (1995).
(This is the reason we described the running time in step-2 of our junction tree algorithm as only $O(N^2)$)

*Remark 4:* Elimination orderings which do not add any edges are called 'perfect elimination orderings'. They have connections to many important problems such as graph coloring, and form an interesting topic of study. It should be clear by now that a graph has a perfect elimination ordering iff it is chordal.

*Remark 5:* The above discussion also shows why the maximum clique problem, which is NP-hard on general graphs, is easy on chordal graphs.

# 5 Definitions of tree graphs

There are several equivalent ways of defining a tree graph. We list some of these definitions below, without giving the proof of equivalence:

*Note:* 1) When we talk about trees, we implicitly talk about spanning trees i.e. trees that connect all the given vertices.
2) In the below definitions, we assume that we are given a set of vertices $V$, where $|V| = N$, and we are looking at graphs over these vertices.

The following are equivalent to the statement "$G$ is a tree" :

**Definition 2** *$G$ is a connected, acyclic graph over N nodes.*

**Definition 3** *$G$ is a connected graph over N nodes with N-1 edges.*

**Definition 4** *$G$ is a* minimal *connected graph over N nodes.*

**Definition 5** *(Important) $G$ is a graph over N nodes, such that for any 2 nodes $i$ and $j$ in $G$, with $i \neq j$, there is a unique path from $i$ to $j$ in $G$.*

## 6 Relationship between chordal graphs and junction trees

**Theorem 2** *For any graph $G = (V, E)$, the following statements are equivalent:*
*1) $G$ has a junction tree.*
*2) $G$ is chordal.*

In order to prove this statement, it will be helpful to recount the definition of junction trees:

**Definition 6 (Junction trees)** *Given a graph $G = (V, E)$, a graph $G' = (V', E')$ is said to be a* Junction Tree *for $G$, iff:*
*i. The nodes of $G'$ are the maximal cliques of $G$ (i.e. $G'$ is a clique graph of $G$.)*
*ii. $G'$ is a tree.*
*iii.* Running Intersection Property / Junction Tree Property: *For each $v \in V$, define $G'_v$ to be the induced subgraph[4] of $G'$ consisting of exactly those nodes which correspond to maximal cliques of $G$ that contain $v$. Then $G'_v$ must be a connected graph.*

We shall now proceed with the proof of the above theorem.

*Proof:*
We need to prove 2 directions:

1. $G$ is chordal $\rightarrow$ $G$ has a junction tree.

This is a proof by induction, and uses a few lemmas about chordal graphs. The full proof is given in the lecture notes for lecture-14, hence we won't repeat it here.

2. $G$ has a junction tree $\rightarrow$ $G$ is chordal.

We prove this by contradiction. Suppose there exists a graph $G$, which is non-chordal, and has a junction tree $G_J$. Since $G$ is not a chordal graph, it must, by definition, have some non-chordal cycle of length 4 or more. For simplicity, we look at the case where the graph has a chordless 4-cycle. The proof for this case generalises easily to chordless k-cycle. WLOG, let vertices 1,2,3,4 be the vertices involved in the chordless 4-cycle in $G$, in that order.

*Note:* To avoid confusion, in what follows, we shall call the nodes of $G$ as 'vertices', and the nodes of $G_J$ as just 'nodes'.

**Claim 3** *The pairs of vertices $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, and $\{4, 1\}$ appear in different nodes of the junction tree.*

*Proof:*
Let us consider the pairs of vertices $\{1, 2\}$ and $\{2, 3\}$. Since $(1, 2)$ is an edge in the original graph, it is either a maximal clique itself or part of some larger maximal clique in $G$. Hence, vertices $\{1, 2\}$ must appear together in some maximal clique of $G$. Similar arguments hold for the other 3 pairs of vertices. However, vertices $\{1, 2, 3\}$ cannot be part of the same clique, since that would mean 1 and 3 are connected, which violates our assumption that (1,2,3,4) is a chordless cycle. Hence, $\{1, 2\}$ and $\{2, 3\}$ must appear in different maximal cliques. By similar arguments, each of the 4 pairs of vertices must appear in different maximal cliques in $G$, and hence in different nodes of $G_J$. This

[4] An induced subgraph is a special type of subgraph where we choose some subset of vertices of the original graph, and ALL edges incident on those vertices.

proves the claim.
■

Let $A$, $B$, $C$, $D$ be some maximal cliques[5] of $G$ containing the pairs of vertices $\{1,2\}$, $\{2,3\}$, $\{3,4\}$, and $\{4,1\}$ respectively. The above claim shows that $A$, $B$, $C$, $D$ must be distinct nodes. Since $G_J$ is a junction tree, by applying the junction tree property to vertex 2, we see that $G_J$ must have some path from A to B where every maximal clique appearing in the path includes vertex 2. By definition, $B$ also contains vertex 3. On the path from A to B, we see if there are any cliques appearing before B, which also contain vertex 3 (this will happen only if $\{2,3\}$ appears in multiple maximal cliques in $G$). We look at the first node in the path which contains vertex 3, call it $B'$. Thus, no intermediate nodes in the path from $A$ to $B'$ contain vertex 3.

Now, we apply the junction tree property to vertex 3. Thus, there must exist a path from $B'$ to $C$, such that every node in the path contains vertex 3. Thus, the path from $B'$ to $C$ cannot have any common nodes with the path from $A$ to $B'$, other than $B'$ (Why?). As in the previous step, we look at the path from $B'$ to $C$, and find the first node in that path that contains vertex 4. Call this node $C'$. Thus, no intermediate node on the path from $B'$ to $C'$ contains vertex 4. We now apply the junction tree property to vertex 4, and thus argue that there exists a path in $G_J$ from $C$ to $D$ such that each node in the path contains vertex 4. Also, this path has no nodes in common with the path from $B'$ to $C'$, other than $C'$.

We are almost done with the proof. We found paths of nodes in $G_J$ from $A$ to $B'$, $B'$ to $C'$, and $C'$ to $D'$, such that the paths were all disjoint except for the endpoints. Hence, putting them one after another, we get a path from $A$ to $D$. We now ask the question - does every node in this path contain vertex 1? The answer is No, because the path contains nodes $B'$ and $C'$, which contain the pairs of vertices $\{2,3\}$ and $\{3,4\}$ respectively, and hence cannot contain vertex 1. (Since we know that vertex 1 is not connected to vertex 3, since (1,2,3,4) is a chordless cycle in $G$). Thus, we have found a path from node $A$ to node $D$ in $G_J$, such that vertex 1 does not appear in every node along the path.

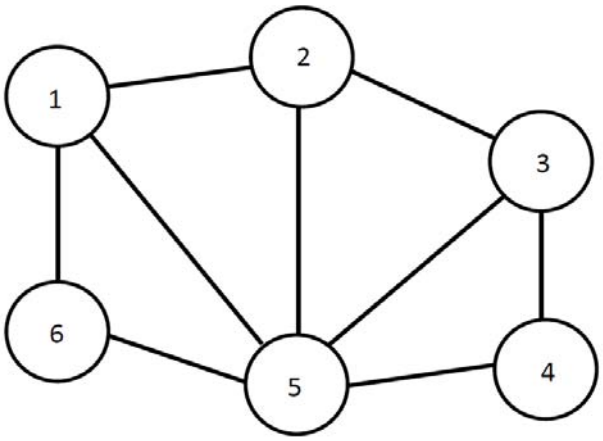
But we know that in a tree, there is a unique path from every node to every other node (see definition-4 for trees in the previous section). Hence, we can now make the following claim - there is No path in $G_J$ from $A$ to $D$, such that vertex 1 appears in all nodes along the path. This means that $G_J$ cannot satisfy the junction tree property with respect to vertex 1. Hence $G_J$ is not a junction tree. QED.
■

# 7 Solutions

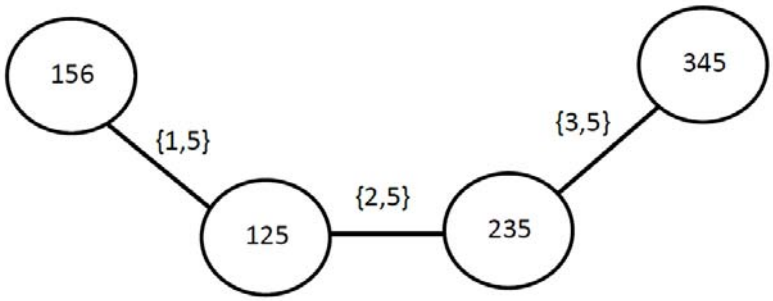
## 7.1 Solution to Junction tree problem

Choose an elimination ordering, say: 6, 1, 2, 3, 4, 5. This ordering adds 2 edges. Here is the triangulated graph:

[5] We are going to use the symbols $A$, $B$, $C$, $D$ to refer to maximal cliques in $G$, as well as nodes in $G_J$.




We now build a junction tree for the chordal graph above. To do this, we first need to build a weighted clique graph, and then find the maximum-weight spanning tree of that graph. Following through with the steps, we get the following junction tree:




Having built the junction tree graph, now we need to assign potentials to it. This is easy to do. For edges, the potentials are merely indicator variables which enforce consistency among adjacent cliques. For nodes, the potentials are the maximal clique potentials corresponding to the triangulated graph. The only thing we need to be careful about is that we do not repeat potentials i.e. we do not assign the same potential in the original graph to multiple maximal cliques in the triangulated graph. This can be done as follows: Maintain a list of potentials over all maximal cliques of the triangulated graph. Initialize all the potentials to one. Now go over each potential in the original graph, and assign it to *some* maximal clique in the triangulated graph. At the end, the potential for each maximal clique is simply the product of the potentials assigned to it.

---

# 6.438 Algorithms for Inference

Fall 2014