14 The Junction Tree Algorithm

In the past few lectures, we looked at exact inference on trees over discrete random variables using Sum-Product and Max-Product, and for trees over multivariate Gaussians using Gaussian belief propagation. Specialized to hidden Markov models, we related Sum-Product to the forward-backward algorithm, Max-Product to the Viterbi algorithm, and Gaussian belief propagation to Kalman filtering and smoothing.

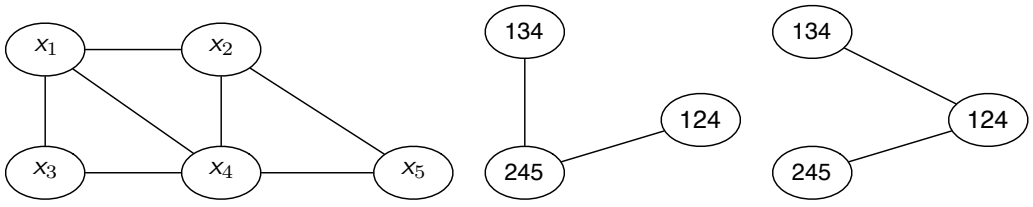
We now venture back to general undirected graphs potentially with loops and again ask how to do exact inference. Our focus will be on computing marginal distributions. To this end, we could just run the Elimination Algorithm to obtain the marginal at a single node and then repeat this for all other nodes in the graph to find all the marginals. But as in the Sum-Product algorithm, it seems that we should be able to recycle some intermediate calculations. Today, we provide an answer to this bookkeeping with an algorithm that does exact inference on general undirected graphs referred to as the *Junction Tree Algorithm*.

At a high-level, the basic idea of the Junction Tree Algorithm is to convert the input graph into a tree and then apply Sum-Product. While this may seem to good to be true, alas, there's no free lunch and the fineprint is that the nodes in the new tree may have alphabet sizes substantially larger than those of the original graph! In particular, we require the trees to be what are called *junction trees*.

14.1 Clique trees and junction trees

The idea behind junction trees is that certain probability distributions corresponding to possibly loopy undirected graphs can be reparameterized as trees, enabling us to run Sum-Product on this tree reparameterization and rest assured that we extract exact marginals.

Before defining what junction trees are, we define *clique graphs* and *clique trees*. Given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a clique graph of $\mathcal{G}$ is a graph where the set of nodes is precisely the set of maximal cliques in $\mathcal{G}$. Next, any clique graph that is a tree is a clique tree. For example, Figures 1b and 1c show two clique trees for the graph in Figure 1a.



(a) (b) (c)

Figure 1: (a) Original graph. (b)-(c) Clique trees for the original graph in (a).

Note that if each x_i takes on a value in $\mathcal{X}$, then each clique node in the example clique trees corresponds to a random variable that takes on $\mathcal{X}^3$ values since all the maximal clique sizes are 3. In other words, clique nodes are supernodes of multiple random variables in the original graph.

Intuitively, the clique tree in Figure 1b seems inadequate for describing the underlying distribution because maximal cliques $\{1,3,4\}$ and $\{1,2,4\}$ both share node 1 but there is no way to encode a constraint that says that the x_1 value that these two clique nodes take on must be the same; we can only put constraints as edge potentials for edges $(\{1,3,4\},\{2,4,5\})$ and $(\{2,4,5\},\{1,2,4\})$. Indeed, what we need is that the maximal cliques that node 1 participates in must form a connected subtree, meaning that if we delete all other maximal cliques that do not involve node 1, then what we are left with should be a connected tree.

With this guiding intuition, we define the *junction tree property*. Let $\mathcal{C}_v$ denote the set of all maximal cliques in the original graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that contain node v. Then we say that a clique graph for $\mathcal{G}$ satisfies the junction tree property if for all $v \in \mathcal{G}$, the set of nodes $\mathcal{C}_v$ in the clique graph induces a connected subtree. For example, the clique tree in Figure 1c satisfies the junction tree property.

Finally, we define a junction tree to be a clique tree that satisfies the junction tree property. Thus, the clique tree in Figure 1c is a junction tree. With our previous intuition for why we need the junction tree property, we can now specify constraints on shared nodes via pairwise potentials. We will first work off our example before discussing the general case.

The original graph in Figure 1a has corresponding factorization

$$p_{x_1,x_2,x_3,x_4,x_5}(x_1,x_2,x_3,x_4,x_5) \propto \psi_{134}(x_1,x_3,x_4)\psi_{124}(x_1,x_2,x_4)\psi_{245}(x_2,x_4,x_5), \quad (1)$$

whereas the junction tree in Figure 1c has corresponding factorization

$$\begin{aligned} &p_{y_{134},y_{124},y_{245}}(y_{134},y_{124},y_{245}) \\ &\quad \propto \phi_{134}(y_{134})\phi_{124}(y_{124})\phi_{245}(y_{245})\psi_{134,124}(y_{134},y_{124})\psi_{124,245}(y_{124},y_{245}). \end{aligned} \quad (2)$$

For notation, we'll denote $[y_V]_S$ to refer to the value that y_V assigns to nodes in S. For example, $[y_{134}]_3$ refers to the value that $y_{134} \in \mathcal{X}^3$ assigns to node 3. Then note that we can equate distributions (1) and (2) by assigning the following potentials:

$$\begin{aligned} \phi_{134}(y_{134}) &= \psi_{134}([y_{134}]_1, [y_{134}]_3, [y_{134}]_4) \\ \phi_{124}(y_{124}) &= \psi_{124}([y_{124}]_1, [y_{124}]_2, [y_{124}]_4) \\ \phi_{245}(y_{245}) &= \psi_{245}([y_{245}]_2, [y_{245}]_4, [y_{245}]_5) \\ \psi_{134,124}(y_{134},y_{124}) &= \mathbf{1}\{[y_{134}]_1 = [y_{124}]_1 \text{ and } [y_{134}]_4 = [y_{124}]_4\} \\ \psi_{124,245}(y_{124},y_{245}) &= \mathbf{1}\{[y_{124}]_2 = [y_{245}]_2 \text{ and } [y_{124}]_4 = [y_{245}]_4\} \end{aligned}$$

The edge potentials constrain shared nodes to take on the same value, and once this consistency is ensured, the rest of the potentials are just the original clique potentials.

In the general case, consider a distribution for graph $\mathcal{G}$ given by

$$p_{\mathbf{x}}(\mathbf{x}) \propto \prod_{C \in \mathcal{C}} \phi_C(x_C), \tag{3}$$

where $\mathcal{C}$ is the set of maximal cliques in $\mathcal{G}$. Then a junction tree $\mathcal{T} = (\mathcal{C}, \mathcal{F})$ for $\mathcal{G}$ has factorization

$$p_{\mathbf{y}}(\{y_C : C \in \mathcal{C}\}) \propto \prod_{C \in \mathcal{C}} \phi_C(y_C) \prod_{U,W \in \mathcal{F}} \psi_{V,W}(y_V, y_W), \tag{4}$$

where edge potential $\psi_{V,W}$ is given by

$$\psi_{V,W}(y_V, y_W) = \mathbf{1}\{[y_V]_S = [y_W]_S\} \qquad \text{for } S = V \cap W.$$

Comparing (3) with (1) and (4) with (2), note that importantly, y_C's are values over cliques. In particular, for $V, W \in \mathcal{C}$ with $S = V \cap W \neq \varnothing$, we can plug into the joint distribution (4) values for y_V and y_W that disagree over shared nodes in S; of course, the probability of such an assignment will just be 0.

With the singleton and edge potentials defined, applying Sum-Product directly yields marginals for all the maximal cliques. However, we wanted marginals for nodes in the original graph, not marginals over maximal cliques! Fortunately, we can extract node marginals from maximal-clique marginals by just some additional marginalization. In particular, for node v in the original graph, we just look at the marginal distribution for any maximal clique that v participates in and sum out the other variables in this maximal clique's marginal distribution.

So we see that junction trees are useful as we can apply Sum-Product on them along with some additional marginalization to compute all the node marginals. But a few key questions arise that demand answers:

1. What graphs over distributions have junction trees?
2. For a graph that has a junction tree, while we know a junction tree exists for it, how do we actually find it?
3. For a graph that does not have a junction tree, how do we modify it so that it does have a junction tree?

We'll answer these in turn, piecing together the full Junction Tree Algorithm.

14.2 Chordal graphs and junction trees

To answer the first question raised, we skip to the punchline:

Theorem 1. *If a graph is chordal, then it has a junction tree.*

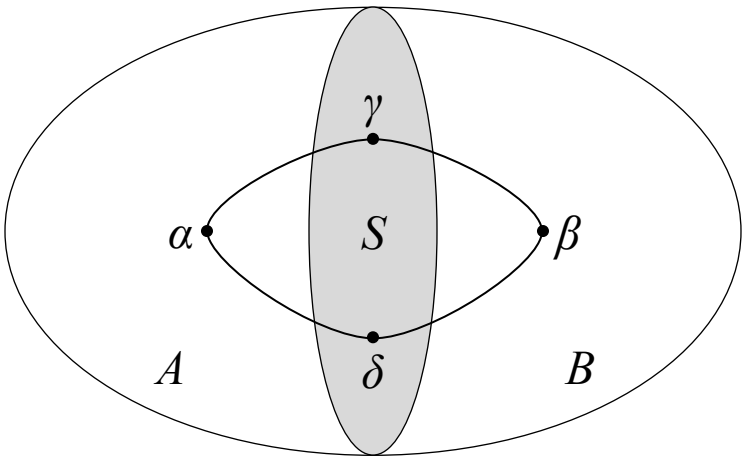


Figure 2: Diagram to help explain the proof of Lemma 1.

In fact, the converse is also true: If a graph has a junction tree, then it must be chordal. We won't need this direction though and will only prove the forward direction. First, we collect a couple of lemmas.

Lemma 1. *If graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is chordal, has at least three nodes, and is not fully connected, then $\mathcal{V} = \mathcal{A} \cup \mathcal{B} \cup \mathcal{S}$ where: (i) sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}$ are disjoint, (ii) $\mathcal{S}$ separates $\mathcal{A}$ from $\mathcal{B}$, and (iii) $\mathcal{S}$ is fully connected.*

Proof. Refer to the visual aid in Figure 2. Nodes $\alpha, \beta \in \mathcal{V}$ are chosen to be nonadjacent (requiring the graph to have at least three nodes and not be fully connected), and $\mathcal{S}$ is chosen to be the minimal set of nodes for which any path from α to β passes through. Define $\mathcal{A}$ to be the set of nodes reachable from α with $\mathcal{S}$ removed, and define $\mathcal{B}$ to be the set of nodes reachable from β with $\mathcal{S}$ removed. By construction, $\mathcal{S}$ separates $\mathcal{A}$ from $\mathcal{B}$ and $\mathcal{V} = \mathcal{A} \cup \mathcal{B} \cup \mathcal{S}$ where $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}$ are disjoint. It remains to show (*iii*).

Let $\gamma, \delta \in \mathcal{S}$. Because $\mathcal{S}$ is the minimal set of nodes for which any path from α to β passes through, there must be a path from α to γ as well as a path from α to δ, from which we conclude that there must be a path from γ to δ in $\mathcal{A} \cup \mathcal{S}$. Similarly, there must be a path from β to γ as well as a path from β to δ, from which we conclude that there must be a path from γ to δ in $\mathcal{B} \cup \mathcal{S}$.

Suppose for contradiction that there is no edge from γ to δ. Consider the shortest path from γ to δ in $\mathcal{A} \cup \mathcal{S}$ and the shortest path from γ to δ in $\mathcal{B} \cup \mathcal{S}$. String these two shortest paths together to form a cycle of at least four nodes. Since we chose shortest paths, this cycle has no chord, which is a contradiction because $\mathcal{G}$ is chordal. Conclude then that there must be an edge from γ to δ. Iterating this argument over all such $\gamma, \delta \in \mathcal{S}$, we conclude that $\mathcal{S}$ is fully connected. □

Lemma 2. *If graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is chordal and has at least two nodes, then $\mathcal{G}$ has at least two nodes each with all its neighbors connected. Furthermore, if $\mathcal{G}$ is not fully connected, then there exist two nonadjacent nodes each with all its neighbors connected.*

Proof. We use strong induction on the number of nodes in the graph. The base case of two nodes is trivial as each node has only one neighbor.

Now for the inductive step: Our inductive hypothesis is that any graph of size up to N nodes which is chordal and has at least two nodes contains at least two nodes each with all its neighbors connected; moreover, if said graph is fully connected then there exist two nonadjacent nodes each with all its neighbors connected.

We want to show that the same holds for any graph $\mathcal{G}'$ over $N+1$ nodes which is chordal and has at least two nodes. Note that if $\mathcal{G}'$ is fully connected, then there is nothing to show since each node has all its neighbors connected, so we can arbitrarily pick any two nodes. The interesting case is when $\mathcal{G}'$ is not fully connected, for which we need to find two nonadjacent nodes each with all its neighbors connected. To do this, we first apply Lemma 1 to decompose the graph into disjoint sets $\mathcal{A}, \mathcal{B}, \mathcal{S}$ with $\mathcal{S}$ separating $\mathcal{A}$ and $\mathcal{B}$. Note that $\mathcal{A} \cup \mathcal{S}$ and $\mathcal{B} \cup \mathcal{S}$ are each chordal with at least two nodes and at most N nodes. The idea is that we will show that we can pick one node from $\mathcal{A}$ that has all its neighbors connected and also one node from $\mathcal{B}$ that has all its neighbors connected, at which point we'd be done. The proof for finding each of these nodes is the same so we just need to show how to find, say, one node from $\mathcal{A}$ that has all its neighbors connected.

By the inductive hypothesis, $\mathcal{A} \cup \mathcal{S}$ has at least two nodes each with all its neighbors connected. In particular, if $\mathcal{A} \cup \mathcal{S}$ is fully connected, then we can just choose any point in $\mathcal{A}$ and its neighbors will all be connected and it certainly won't have any neighbors in $\mathcal{B}$ since $\mathcal{S}$ separates $\mathcal{A}$ from $\mathcal{B}$, at which point we're done. On the other hand, if $\mathcal{A} \cup \mathcal{S}$ is not fully connected, by the inductive hypothesis, there exist two nonadjacent nodes in $\mathcal{A} \cup \mathcal{S}$ each with all its neighbors connected, for which certainly one will be $\mathcal{A}$ because if both are in $\mathcal{S}$, they would actually be adjacent (since $\mathcal{S}$ is fully connected). □

We now prove Theorem 1:

Proof. We use induction on the number of nodes in the graph. For the base case, we note that if a chordal graph has only one or two nodes, then trivially the graph has a junction tree.

Onto the inductive step: Suppose that any graph of up to N nodes that is chordal has a junction tree. We want to show that chordal graph $\mathcal{G}$ with $N+1$ nodes also has a junction tree. By Lemma 2, $\mathcal{G}$ has a node α with all its neighbors connected. This means that removing node α from $\mathcal{G}$ will result in a graph $\mathcal{G}'$ which is also chordal. By the inductive hypothesis, $\mathcal{G}'$, which has N nodes, has a junction tree $\mathcal{T}'$. We next show that we can modify $\mathcal{T}'$ to obtain a new junction tree $\mathcal{T}$, which is the junction tree for our $(N+1)$-node chordal graph $\mathcal{G}$.

Let C be the maximal clique that node α participates in, which consists of α and its neighbors, which are all connected. If $C \setminus \alpha$ is a maximal-clique node in $\mathcal{T}'$, then we can just add α to this clique node to obtain a junction tree $\mathcal{T}$ for $\mathcal{G}$.

Otherwise, if $C \setminus \alpha$ is not a maximal-clique node in $\mathcal{T}'$, then $C \setminus \alpha$, which we know to be a clique, must be a subset of a a maximal-clique node D in $\mathcal{T}'$. Then we add C as a new maximal-clique node in $\mathcal{T}'$ which we connect to D to obtain a junction tree $\mathcal{T}$ for $\mathcal{G}$; justifying why $\mathcal{T}$ is in fact a junction tree just involves checking the junction tree property and noting that α participates only in maximal clique C. □

14.3 Finding a junction tree for a chordal graph

We now know that chordal graphs have junction trees. However, the proof given for Theorem 1 is existential and fails to deliver an efficient algorithm. Recalling our second question raised from before, we now ask how we actually construct a junction tree given a chordal graph.

Surprisingly, finding a junction tree for a chordal graph just boils down to finding a maximum-weight spanning tree, which can be efficiently solved using Kruskal's algorithm or Prim's algorithm. More precisely:

Theorem 2. *Consider weighted graph $\mathcal{H}$, which is a clique graph for some underlying graph $\mathcal{G}$ where we have an edge between two maximal-clique nodes V, W with weight $|V \cap W|$ whenever this weight is positive.*

Then a clique tree for underlying graph $\mathcal{G}$ is a junction tree if and only if it is a maximum-weight spanning tree in $\mathcal{H}$.

Before we prove this, we state a key inequality: Let clique tree $\mathcal{T} = (\mathcal{C}, \mathcal{F})$ be for underlying graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with maximal cliques $\mathcal{C}$. Denote S_e as the separator set for edge $e \in \mathcal{F}$, i.e., S_e is the nodes in common between the two maximal cliques connected by e. Then any $v \in \mathcal{V}$ satisfies:

$$\sum_{e \in \mathcal{F}} \mathbf{1}\{v \in S_e\} \leq \sum_{C \in \mathcal{C}} \mathbf{1}\{v \in C\} - 1. \tag{5}$$

What this is saying is that the number of separator sets v participates in is bounded above by the number of maximal cliques v participates in minus 1. The intuition for why there's a minus 1 is that the left-hand side is counting edges in the tree and the right-hand side is counting nodes in the tree (remember that a tree with N nodes has $N - 1$ edges). As a simple example, if v participates in two maximal cliques, then v will participate in at most one separator set, which is precisely the intersection of the two maximal cliques.

With the above intuition, note that the inequality above becomes an equality if and only if $\mathcal{T}$ is a junction tree! To see this, after dropping terms on both sides that have nothing to do with node v, the left-hand side counts edges that v is associated with and the right-hand side counts maximal-clique nodes that v is associated with

minus 1. So equality holds when $\mathcal{T}$ restricted to maximal cliques that v participates in forms a connected subtree.

Now we prove Theorem 2:

Proof. Let $\mathcal{T} = (\mathcal{C}, \mathcal{F})$ be a clique tree for underlying graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Note that $\mathcal{T}$ is a subgraph of weighted graph $\mathcal{H}$. Denote $w(\mathcal{T})$ to be the sum of all the weights along edges of $\mathcal{T}$ using the edge weights assigned. We find an upper-bound for $w(\mathcal{T})$, using inequality (5):

$$\begin{aligned}
w(\mathcal{T}) &= \sum_{e \in \mathcal{F}} |S_e| \\
&= \sum_{e \in \mathcal{F}} \sum_{v \in \mathcal{V}} \mathbf{1}\{v \in S_e\} \\
&= \sum_{v \in \mathcal{V}} \sum_{e \in \mathcal{F}} \mathbf{1}\{v \in S_e\} \\
&\leq \sum_{v \in \mathcal{V}} \left(\sum_{C \in \mathcal{C}} \mathbf{1}\{v \in C\} - 1 \right) \\
&= \sum_{v \in \mathcal{V}} \sum_{C \in \mathcal{C}} \mathbf{1}\{v \in C\} - |V| \\
&= \sum_{C \in \mathcal{C}} \sum_{v \in \mathcal{V}} \mathbf{1}\{v \in C\} - |V| \\
&= \sum_{C \in \mathcal{C}} |C| - |V|.
\end{aligned}$$

This means that any maximum-weight spanning (clique) tree $\mathcal{T}$ for $\mathcal{H}$ has weight bounded above by $\sum_{C \in \mathcal{C}} |C| - |V|$, where equality is attained if and only if inequality (5) is an equality, i.e., when $\mathcal{T}$ is a junction tree. □

14.4 The full algorithm

The last question asked was that if a graph doesn't have a junction tree, how can we modify it so that it does have a junction tree? This question was actually already answered when we discussed the Elimination Algorithm. In particular, when running the Elimination Algorithm, the reconstituted graph is always chordal. So it suffices to run the Elimination Algorithm using any elimination ordering to obtain the (chordal) reconstituted graph.

With this last question answered, we outline the key steps for the Junction Tree Algorithm. As input, we have an undirected graph and an elimination ordering, and the output comprises of all the node marginals.

1. Chordalize the input graph using the Elimination Algorithm.

2. Find all the maximal cliques in the chordal graph.
3. Determine all the separator set sizes for the maximal cliques to build a (possibly loopy) weighted clique graph.
4. Find a maximum-weight spanning tree in the weighted clique graph from the previous step to obtain a junction tree.
5. Assign singleton and edge potentials to the junction tree.
6. Run Sum-Product on the junction tree.
7. Do additional marginalization to get node marginals for each node in the original input graph.

As with the Elimination Algorithm, we incorporate observations by fixing observed variables to take on specific values as a preprocessing step. Also, we note that we've intentionally brushed aside implementation details, which give rise to different variants of the Junction Tree Algorithm that have been named after different people. For example, by accounting for the structure of the edge potentials, we can simplify the Sum-Product message passing equation, resulting in the Shafer-Shenoy algorithm. Additional clever bookkeeping to avoid having to multiply many messages repeatedly yields the Hugin algorithm.

We end by discussing the computational complexity of the Junction Tree Algorithm. The key quantity of interest is the treewidth of the original graph, defined in a previous lecture as the largest maximal clique size for an optimal elimination ordering. Note that using an optimal elimination ordering will result in the largest maximal-clique node in the junction tree having alphabet size exponential in the treewidth, so the Junction Tree Algorithm will consequently have running time exponential in the treewidth.

Yet, the fact that the Junction Tree Algorithm may take exponential time should not be surprising. Since we can encode NP-complete problems such as 3-coloring as an inference problem over an undirected graph, if the Junction Tree Algorithm could efficiently solve all inference problems over undirected graphs, then the P vs. NP debate would have long ago been settled.

6.438 Algorithms for Inference

Fall 2014