IFT 6269: Probabilistic Graphical Models **Fall 2020**

Lecture 16 — November 3

Lecturer: Simon Lacoste-Julien *Scribe: Ismael Martinez, Abdelrahman Zayed*

Disclaimer: Lightly proofread and quickly corrected by Simon Lacoste-Julien.

16.1 Inference on trees

To do inference on a tree, we can use the graph eliminate algorithm as described in the last lecture, using an appropriate elimination ordering as exemplified in Fig. 16.1. This corresponds to marginalizing (16.1), using the distributivity trick. A good order to perform graph elimination on a tree is to eliminate the **leaves first** (which makes sure that no new edges are added in the augmented graph, achieving the treewidth of one (maximal clique size of 2)).

$$p(x) = \frac{1}{Z} \prod_{i=1}^{n} \psi_i(x_i) \prod_{\{i,j\} \in E}^{n} \psi_{i,j}(x_i, x_j) \tag{16.1}$$

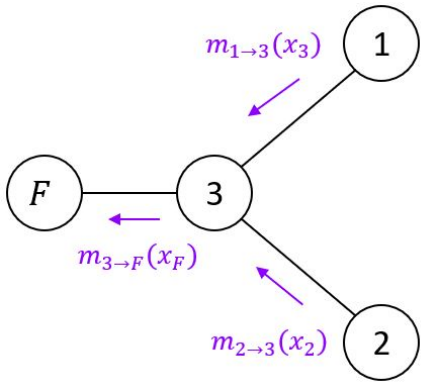


Figure 16.1: Applying graph elimination to compute $p(x_F)$.

We can also apply the graph elimination by using F as a root as shown in Fig.16.2. The messages that are passed from leaf nodes towards the root are computed according to (16.2).

$$m_{i\to j}(x_j) = \sum_{x_i} \psi_i(x_i)\psi_{i,j}(x_i, x_j) \underbrace{\prod_{k \in children(i)} m_{k\to i}(x_i)}_{\text{new factors containing } i \text{ on active list}} \tag{16.2}$$

where node i is the child of node j.

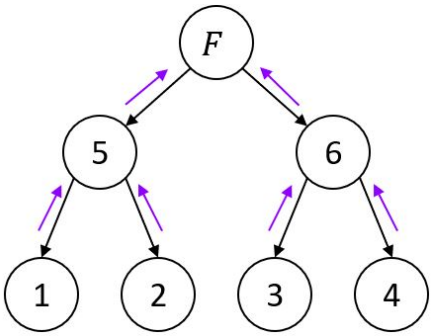


Figure 16.2: Using F as the root node to compute $p(x_F)$

16.1.1 Sum-product algorithm for trees

The sum-product algorithm (SPA) is an algorithm to get all the node/edge marginals cheaply by storing (caching) and reusing the messages using dynamic programming.

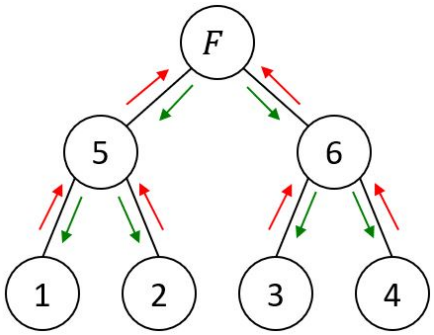


Figure 16.3: The collect and distribute phases to compute the marginal of any node. The red arrows refer to the collect phase, whereas the green arrows refer to the distribute phase.

The message from node i to node j is computed as follows:

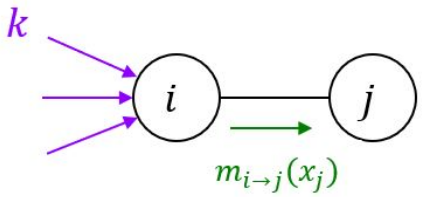


Figure 16.4: Passing the message from node i to node j.

$$m_{i\to j}(x_j) = \sum_{x_i} \psi_j(x_i)\psi_{i,j}(x_i, x_j) \prod_{k\in N(i)\setminus\{j\}} m_{k\to i}(x_i) \tag{16.3}$$

where $N(i)$ is of set of neighbors of i.

By using the green and red arrows as shown in Fig.16.3, we can compute the marginal of any node. The goal is to compute $m_{i\to j}(x_j)$ and $m_{j\to i}(x_i)\ \forall\ \{i,j\} \in E$. Node i can only send messages to its neighbour j when it has received all messages from other neighbours.

The message from node i to node j is computed according to Fig. 16.4. At the end, the node marginal is proportional to all the factors left on the active list, and in this message passing formulation, it is proportional to

$$p(x_i) \propto \prod_{j\in N(i)} m_{j\to i}(x_i)\psi_i(x_i) \tag{16.4}$$

Therefore,

$$Z = \sum_{x_i} \prod_{j\in N(i)} (m_{j\to i}(x_i)\psi_i(x_i)) \tag{16.5}$$

The **edge marginal** probability of a pair of neighboring node i and node j (see Fig. 16.5 for an example) is computed as follows:

$$p(x_i, x_j) = \frac{1}{Z}\psi_i(x_i)\psi_j(x_j)\psi_{i,j}(x_i, x_j) \prod_{k\in N(i)\setminus\{j\}} m_{k\to i}(x_i) \prod_{k\prime\in N(j)\setminus\{i\}} m_{k'\to j}(x_j) \tag{16.6}$$

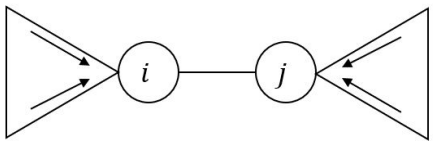


Figure 16.5: Computing the marginal probability of the edge between node i and node j.

If we want to compute the marginal on a pair of nodes which are not adjacent, then we usually need to use the more general graph eliminate algorithm. See Fig. 16.6 for an example. SPA only works to compute all the node marginals, as well as the edge marginals for adjacent nodes.[1]

[1]You could generalize the argument made in Fig. 16.5 to compute the marginals on more than 2 nodes (if they are all connected) by taking the product of all the incoming messages to the boundary of the connected nodes, as well as all the potentials connecting the nodes.

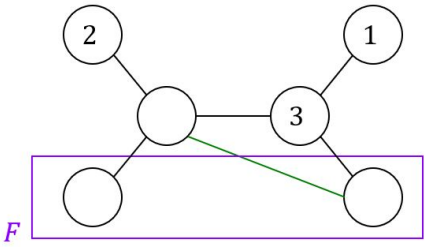


Figure 16.6: Example of a case where we need to use the graph elimination algorithm to compute the probability of the set of nodes in the box. In green is the added edge appearing in the augmented graph when running graph eliminate (there is no node ordering that we can choose with F at the end which would not add edges, explaining why the sum-product algorithm cannot be used for this example).

Sum-product schedule

a) We can use the collect/distribute schedule as shown in Fig. 16.3.

b) We can also use the **flooding parallel** schedule, which works as follows:

1) Initialize all $m_{i\to j}(x_j)$ messages to a uniform distribution $\forall(i,j),(j,i)$ s.t. $\{i,j\}\in E$.
2) At every step (in parallel), compute $m^{\text{new}}_{i\to j}(x_j)$ as if the neighbour messages were correctly computed from previous step.

→ One can prove that for a tree of diameter d, all messages are correctly computed and fixed after d steps (they are fixed points of this update process)

16.1.2 Sum-product algorithm for graphs with cycles – loopy belief propagation

Whereas parallel SPA iteratively computes messages on a tree, *loopy belief propagation* provides the general case for approximate inference for graphs with cycles. *Loopy* refers to cycles.

1. Initialise all $m_{i\to j}(x_j)$ messages (by uniform distribution) $\forall(i,j),(j,i)$ s.t. $\{i,j\}\in E$.
2. At every step, compute $m^{\text{new}}_{j\to i}(x_i)$ using a convex combination (in the log domain) between the previous message and the new calculation to stabilize the update.

$$m^{\text{new}}_{i\to j}(x_j) = (m^{\text{old}}_{i\to j}(x_j))^{\alpha}\left(\sum_{x_i}\psi_i(x_i)\psi_{i,j}(x_i,x_j)\prod_{k\in N(i)\setminus\{j\}}m^{\text{old}}_{k\to i}(x_i)\right)^{1-\alpha} \tag{16.7}$$

where $\alpha\in[0,1]$ step-size. This approach is known as "damping".

Remarks

- This gives exact answer on trees (fixed point → yields correct marginal).
- If G is not a tree, the algorithm doesn't converge in general to the right marginal, but sometimes (if not too loopy) gives reasonable approximations.

Getting conditionals

$$p(x_i|\bar{x}_E) \propto p(x_i, \bar{x}_E) \tag{16.8}$$

The bar is used to indicate the fixed values we are conditioning on. We keep the variables $\bar{x}_E$ fixed during marginalization for each $j \in E$.

(Formal trick): by redefining the potential function, we don't need to worry about fixing the variables. Redefine $\tilde{\psi}_j(x_j) \triangleq \psi_j(x_j) \cdot \delta(x_j, \bar{x}_j)$.

Kronecker-delta function $\delta(a,b) \triangleq \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise.} \end{cases}$

Computing $m_{j\to i}(x_i)$ for $j \in E$:

$$\sum_{x_j} \tilde{\psi}_j(x_j)\text{stuff}(x_j, x_i) = \psi_j(\bar{x}_j)\text{stuff}(\bar{x}_j, x_i). \tag{16.9}$$

At the end, result of SPA will give

$$p(x_i, \bar{x}_E) = \frac{1}{Z}\psi_i(x_i) \prod_{k\to i} m_{k\to i}(x_i) \tag{16.10}$$

Normalize over x_i to get conditional $p(x_i|\bar{x}_E)$.

Lesson: When we run graph-eliminate or SPA, we don't sum over the observed variables.

Note: Sum-product is to compute the marginal; max-product is to compute the arg max!

16.1.3 Max-Product Algorithm

For SPA, main property used was distributivity of $\oplus$ over $\odot$. We require that $(\mathbb{R}, \oplus, \odot)$ is a semi-ring (i.e. don't need additive inverses).

You can do "sum-product" like algorithms on other semi-rings, where we replace the operations but use the same concepts:

$$(\mathbb{R}, \max, +) \qquad \max(a+b, a+c) = a + \max(b, c) \tag{16.11}$$

$$(\mathbb{R}_+, \max, \cdot) \qquad \max(a \cdot b, a \cdot c) = a \cdot \max(b, c) \tag{16.12}$$

The second example above is where we get the max-product algorithm. The distributivity trick that I had mentioned previously to motivate the graph eliminate algorithm, using this max-product semi-ring, takes the form:

$$\max_{x_{1:n}} \prod_i f_i(x_i) = \prod_i \max_{x_i} f_i(x_i).$$

Analogous to SPA where we move the sum from outside the product to inside, we move the max function from outside the product to inside. The message updates for max-product algorithm become

$$m_{i\to j}(x_j) = \max_{x_i} \left[\psi_{i,j}(x_i, x_j)\psi_i(x_i) \prod_{k\in N(i)\setminus\{j\}} m_{k\to i}(x_i) \right]. \tag{16.13}$$

Example For the example in Fig. 16.7, $\max_{x_{1:5}} \frac{1}{Z}\prod \psi_c(x_c) = \frac{1}{Z}\max_{x_1} m_{2\to 1}(x_1)$.

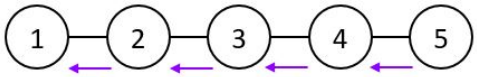


Figure 16.7: Sequential message passing to compute the argmax.

To get the arg max, store argument of this max as a function of x_j for every j. To get $\arg\max p(x_{1:n})$, run max-product algorithm forward, and backtrack the arg max pointers to get the full arg max. This algorithm of "decoding" by backtracking is also known as the Viterbi algorithm (see Fig. 16.8).

Property of tree UGM

Let $p \in \mathcal{L}(G)$, for $G = (V, E)$ a tree with non-zero marginals. Then we have:

$$p(x_{1:n}) = \frac{1}{Z} \prod_{i\in V} \underbrace{p(x_i)}_{\psi_i(x_i)} \prod_{\{i,j\}\in E} \underbrace{\frac{p(x_i, x_j)}{p(x_i)p(x_j)}}_{\psi_{i,j}(x_i,x_j)} \tag{16.14}$$

Proof idea: Similar to DGM, we define a set of factors $\{f_{i,j}(x_i, x_j)\}$, $\{f_i(x_i)\}$, $f_{i,j} \geq 0, f_i \geq 0$ such that "local consistency property" holds:

$$\sum_{x_j} f_{i,j}(x_i, x_j) = f_i(x_i)\ \forall x_i \tag{16.15}$$

$$\sum_{x_i} f_{i,j}(x_i, x_j) = f_j(x_j)\ \forall x_j \tag{16.16}$$

$$\sum_{x_i} f_i(x_i) = 1. \tag{16.17}$$

Then, if we define joint

$$p(x) = \prod_i f_i(x_i) \prod_{\{i,j\}\in E} \frac{f(x_j, x_i)}{f(x_j)f(x_i)}. \tag{16.18}$$

we can show we get the correct marginals, i.e. $p(x_i) = f_i(x_i)$.

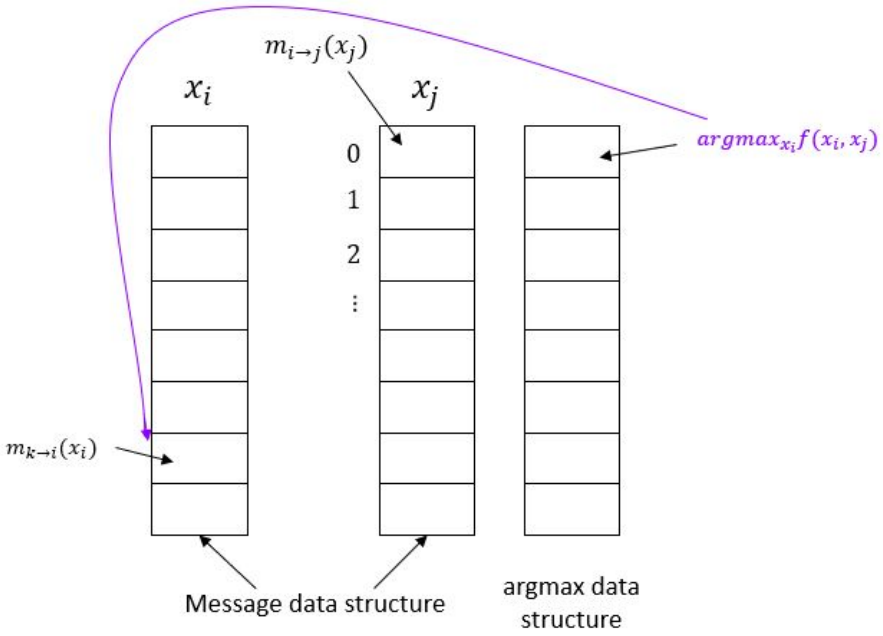


Figure 16.8: Store arg max values in a forward pass, and backtrack to pointers to get all values. This is the "Viterbi" algorithm.

16.1.4 Junction tree algorithm

The junction tree algorithm is an algorithm designed to tackle the problem of *inference on general triangulated graphs*. It is a generalization of SPA to a clique tree with the junction tree property.

Fig. 16.9 show a clique tree with the "running intersection property": if $j \in C_1 \cap C_2$, then $j \in C_k \ \forall C_k$ along the path from C_1 to C_2. A tree that satisfies this property is known as a "junction tree".

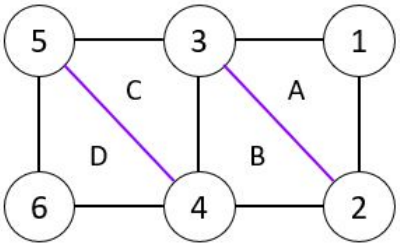
Build a junction tree from triangulated graph

Use maximum weight spanning tree on the clique graph, where the size of separator sets are the weights on the edges in the clique graph. In other words, a spanning tree with the maximum number of nodes in common among neighbouring cliques. This tree will have the running intersection property, and is therefore a junction tree.

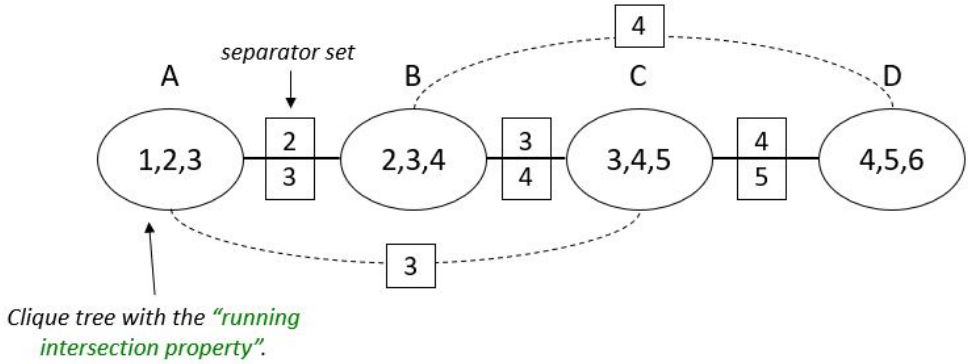
Theorem 16.1 *$\exists$ a junction tree $\iff$ the graph is triangulated graph (decomposable graph).*

You can always turn a graph into a triangulated graph by running graph-eliminate algorithm. Once you have a junction tree, you can show

$$p(x_V) = \frac{\prod_C p(x_C)}{\prod_S p(x_S)} \tag{16.19}$$



(a) Triangulated graph representation.



(b) Clique tree representation.

Figure 16.9: A clique tree that follows the running intersection property is known as a Junction tree. The junction tree is shown here with the non-dotted edges. The dotted edges are additional edges present in the clique graph (where an edge is put between every pair of cliques with some node in common). One can build a spanning tree which has the running intersection property by running the maximum weight spanning tree algorithm on the clique graph with weight on edges that is the size of the separator set.

where S is the separator sets in the junction tree.

Junction tree algorithm :

1. Reconstruct the above formulation by starting with
$$p(x_V) = \frac{1}{Z} \frac{\prod_C \psi_C(x_C)}{\prod_S \varphi_S(x_S)} \tag{16.20}$$
where $\varphi_S(x_S) = 1$ at initialization.
2. Do message passing on junction tree to update the potentials ψ_C^{new} and φ_S^{new}.
3. Repeat step 2 until convergence.

At the end, we will have the correct marginals $p(x_C)$ and $p(x_S)$.