Sum-Product, Max-Sum and Beyond

Bishop §8.4.4 - §8.4.8

Andrew Predoehl

29 January 2009

Outline

1. The sum-product algorithm
   - Essentials
   - Simple Example
   - General formulation
2. The max-sum algorithm
   - Essential idea
   - Implementation
3. Beyond sum-product and max-sum

Outline

1. The sum-product algorithm
   - Essentials
   - Simple Example
   - General formulation
2. The max-sum algorithm
   - Essential idea
   - Implementation
3. Beyond sum-product and max-sum

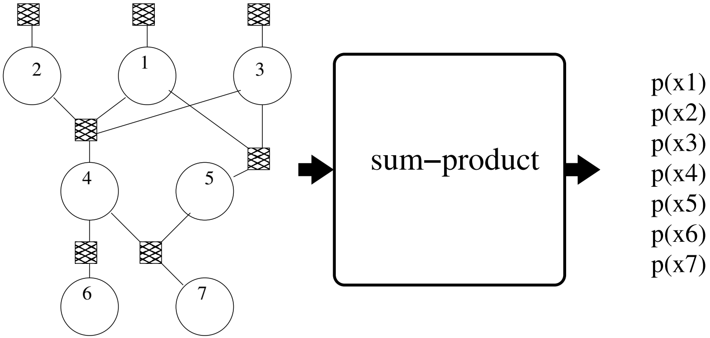
Purpose

The sum-product algorithm lets us do exact inference on factor graphs that are trees.

- “Exact inference” means that we can marginalize any of the variables in the model, i.e., compute $p(x_5)$ from $p(\mathbf{x})$.
- “Tree” means the graph has no cycles (which Bishop calls loops).

Input and Outputs

Input: a factor graph and all the relevant factors
Output: any or all desired marginals

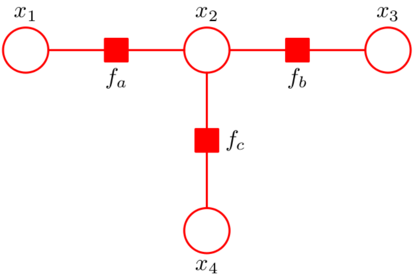


fa(x1), fb(x2), fc(x3),
fd(x1,x2,x3,x4), fe(x1,x3,x5),
fg(x4,x6), fh(x4,x5,x7)

Outline

Key Fact

The algorithm is efficient because the tree topology lets us interchange sums and products. Example: Fig. 8.51 (p. 409)



Ignoring normalization, the joint probability is
$p(x_1, x_2, x_3, x_4) = f_a(x_1, x_2) \cdot f_b(x_2, x_3) \cdot f_c(x_2, x_4)$

For instance, to find $p(x_2)$ we must marginalize over the other variables. This easily becomes a product of sums:

$$
\begin{aligned}
p(x_2) &= \sum_{x_1,x_3,x_4} p(x_1,x_2,x_3,x_4) && (1)\\
&= \sum_{x_4}\left(\sum_{x_3}\left(\sum_{x_1}\left(f_a(x_1,x_2)f_b(x_2,x_3)f_c(x_2,x_4)\right)\right)\right) && (2)\\
&= \sum_{x_4}\left(\sum_{x_3}\left(\sum_{x_1} f_a(x_1,x_2)\right)f_b(x_2,x_3)\cdot f_c(x_2,x_4)\right) && (3)\\
&= \left(\sum_{x_1} f_a(x_1,x_2)\right)\sum_{x_4}\left(\sum_{x_3} f_b(x_2,x_3)\cdot f_c(x_2,x_4)\right) && (4)\\
&\vdots\\
&= \left(\sum_{x_1} f_a(x_1,x_2)\right)\left(\sum_{x_3} f_b(x_2,x_3)\right)\left(\sum_{x_4} f_c(x_2,x_4)\right) && (5)
\end{aligned}
$$

Messages

So we see a *nested sum of products* becomes a *sequential product of N sums*. Time: $O(NK^c)$ instead of $O(K^N)$. The result may be thought of as a product of *messages*.

$$\begin{aligned} p(x_2) &= \Big(\sum_{x_1} f_a(x_1, x_2)\Big)\Big(\sum_{x_3} f_b(x_2, x_3)\Big)\Big(\sum_{x_4} f_c(x_2, x_4)\Big) \\ &= \Big(\mu_{f_a \to x_2}(x_2) \Big)\Big(\mu_{f_b \to x_2}(x_2) \Big)\Big(\mu_{f_c \to x_2}(x_2) \Big) \quad (6) \end{aligned}$$

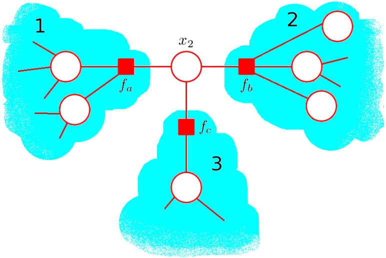
Each message represents how the marginalized variables influence the distribution of the desired variable.

Outline

1. The sum-product algorithm
   - Essentials
   - Simple Example
   - General formulation
2. The max-sum algorithm
   - Essential idea
   - Implementation
3. Beyond sum-product and max-sum

Does this trick always work?

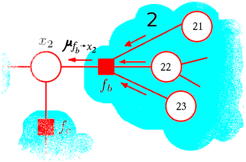
Any factor graph that is a *tree* can use this trick at every node.



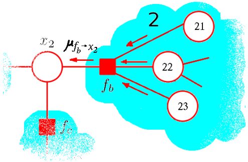
Why? Because of disjoint blobs: every factor in just one blob.

$$p(x_2) = \sum_{1,2,3} \prod_{1,2,3} f_i() = \Big(\sum_{blob_1} \prod_{blob_1} f()\Big)\Big(\sum_{blob_2} \prod_{blob_2} f()\Big)\Big(\sum_{blob_3} \prod_{blob_3} f()\Big)$$

Likewise, each $f \to x$ message can be factored:



$$
\begin{aligned}
\mu_{f_b \to x_2} &= \sum_{blob_2} \prod_{blob_2} f(\dots) && (7)\\
&= \sum_{x_{21}} \sum_{x_{22}} \sum_{x_{23}} \sum_{etc.} f_b(x_2, x_{21}, x_{22}, x_{23}) \prod_{etc.} f(\dots) && (8)\\
&= \sum_{x_{21}} \sum_{x_{22}} \sum_{x_{23}} f_b(x_2, x_{21}, x_{22}, x_{23}) \sum_{etc.} \prod_{etc.} f(\dots) && (9)\\
&= \sum_{x_{21}} \sum_{x_{22}} \sum_{x_{23}} f_b(x_2, x_{21}, x_{22}, x_{23}) \prod_{etc.} \Big(\sum_{etc.} f(\dots) \Big) && (10)
\end{aligned}
$$



We call these factors the messages *from variable to factor node*

$$
\begin{aligned}
\mu_{f_b \to x_2} &= \sum_{x_{21}} \sum_{x_{22}} \sum_{x_{23}} f_b(x_2, x_{21}, x_{22}, x_{23}) \prod_{etc.} \left(\sum_{etc.} f(\ldots) \right) \\
&= \sum_{x_{21}} \sum_{x_{22}} \sum_{x_{23}} f_b(x_2, x_{21}, x_{22}, x_{23}) \prod_{etc.} \left(\mu_{x_k \to f_b} \right) \quad (11)
\end{aligned}
$$

So to compute a single marginal $p(x)$

- Start at the leaves of the tree (considering x as root)
- Compute messages going towards x, using (6) & (11)

So to compute a single marginal $p(x)$

- Start at the leaves of the tree (considering x as root)
- Compute messages going towards x, using (6) & (11)

To compute *all* the marginals, for each edge in the graph,

- Cache the messages going towards x
- Then compute (and cache) all messages in the opposite direction

So to compute a single marginal $p(x)$

- Start at the leaves of the tree (considering x as root)
- Compute messages going towards x, using (6) & (11)

To compute *all* the marginals, for each edge in the graph,

- Cache the messages going towards x
- Then compute (and cache) all messages in the opposite direction

Need normalization? Use $p(x)$ or any other marginal.

Outline

- What if, instead of the marginals, we want the *maximum likelihood* setting of the variables, that is, the particular choice of $\mathbf{x} = \mathbf{x}_{ML}$ such that the joint $p(\mathbf{x}_{ML})$ is maximized?

- What if, instead of the marginals, we want the *maximum likelihood* setting of the variables, that is, the particular choice of $\mathbf{x} = \mathbf{x}_{ML}$ such that the joint $p(\mathbf{x}_{ML})$ is maximized?
- A surprisingly simple modification to sum-product can give us this information.
- The reason it works is that sum and maximum both allow factoring:

$$\sum_{i=0}^{4} a(i-3)^2 = a\sum_{i=0}^{4}(i-3)^2$$

and likewise, if $a > 0$,

$$\max_{i\in\{0,\dots,4\}} a(i-3)^2 = a\left(\max_{i\in\{0,\dots,4\}}(i-3)^2\right).$$

Outline

1. The sum-product algorithm
   - Essentials
   - Simple Example
   - General formulation
2. The max-sum algorithm
   - Essential idea
   - Implementation
3. Beyond sum-product and max-sum

- We do message passing almost the same as before, replacing summations with max operations.
- So you might think of this algorithm as *max-product*; BUT,

- We do message passing almost the same as before, replacing summations with max operations.
- So you might think of this algorithm as *max-product*; BUT,
- As a practical matter, we would hit arithmetic underflow problems; so we use logarithms.
- Thus products of distributions become sums of log probabilities:

$$f_a(x_3, x_4) \cdot f_b(x_3, x_2) \;\longrightarrow\; \ln f_a(x_3, x_4) + \ln f_b(x_3, x_2)$$

- The algorithm is therefore called *max-sum*.

- We do message passing almost the same as before, replacing summations with max operations.
- So you might think of this algorithm as *max-product*; BUT,
- As a practical matter, we would hit arithmetic underflow problems; so we use logarithms.
- Thus products of distributions become sums of log probabilities:

$$f_a(x_3, x_4) \cdot f_b(x_3, x_2) \;\longrightarrow\; \ln f_a(x_3, x_4) + \ln f_b(x_3, x_2)$$

- The algorithm is therefore called *max-sum*.

In this manner we can easily find the maximum probability of the joint distribution, i.e.,

$$p_{ML} \equiv \max_{\mathbf{x}} p(\mathbf{x}).$$

What about argmax?

- To find the $\arg\max$ (the value $\mathbf{x}_{ML}$ at which $p(\mathbf{x}_{ML}) = p_{ML}$), we must perform some extra bookkeeping.

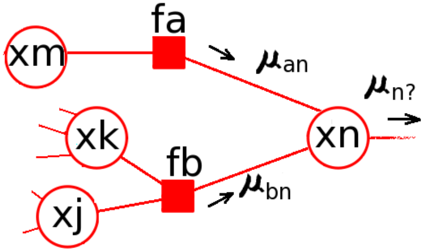
What about argmax?

- To find the arg max (the value $\mathbf{x}_{ML}$ at which $p(\mathbf{x}_{ML}) = p_{ML}$), we must perform some extra bookkeeping.
- Every time we perform a max operation we also store the settings of the adjacent variables that led to the maximum. (Not necessarily unique.)

What about argmax?

- To find the arg max (the value $\mathbf{x}_{ML}$ at which $p(\mathbf{x}_{ML}) = p_{ML}$), we must perform some extra bookkeeping.
- Every time we perform a max operation we also store the settings of the adjacent variables that led to the maximum. (Not necessarily unique.)
- Remember that messages are (in spirit) functions, not values! We need the max for every possible value of the variable.
- For discrete random variables taking one of K possible values, that means we need to store a table of K settings with each variable node, storing the childrens' settings for each value.

Max-sum bookkeeping (with fake numbers)



xn	μ_{an}	xm	μ_{bn}	xj xk
0	-34	2	-26	0,3
1	-16	2	-97	1,3
2	-73	3	-41	2,1
3	-22	0	-60	2,0

$$\mu_{an}(x_n) = \max_{x_m}(\ln f_a(x_n, \dots) + \mu_{ma})$$
$$\mu_{bn}(x_n) = \max_{x_j, x_k}(\ln f_b(x_n, \dots) + \mu_{jb} + \mu_{kb})$$
$$\mu_{n?}(x_n) = \mu_{an}(x_n) + \mu_{bn}(x_n)$$

Wrapping up exact inference

- Once we find the joint max probability, we then backtrack to find a set of variable settings that achieves this value.
- The Viterbi algorithm is a famous example of this: it finds the argmax for a chain of hidden variables.

Wrapping up exact inference

- Once we find the joint max probability, we then backtrack to find a set of variable settings that achieves this value.
- The Viterbi algorithm is a famous example of this: it finds the argmax for a chain of hidden variables.
- When some variables are observed, we can just substitute the observed value for the variable (eliminating a max operation).
- A similar idea applies for sum-product: fix each observed variable to its known value, eliminate a sum.

What do we do with non-tree factor graphs?

Non-tree graphs can be solved too, exactly or approximately:

- There's the *junction tree algorithm* which solves the problem.
- It sounds slow (exponential cost, p. 417).

What do we do with non-tree factor graphs?

Non-tree graphs can be solved too, exactly or approximately:

- There's the *junction tree algorithm* which solves the problem.
- It sounds slow (exponential cost, p. 417).

If approximate results are adequate, there are many approaches:

- Monte Carlo methods (a.k.a. Sampling): very important

What do we do with non-tree factor graphs?

Non-tree graphs can be solved too, exactly or approximately:

- There's the *junction tree algorithm* which solves the problem.
- It sounds slow (exponential cost, p. 417).

If approximate results are adequate, there are many approaches:

- Monte Carlo methods (a.k.a. Sampling): very important
- "Loopy Belief Propagation": use sum-product on the graph, and hope it converges.

Summary

- Sum-product lets you find marginal distribution of variables.
- Max-sum lets you find the ML variable settings.
- These algorithms require the factor graph to be a tree.
- If there are cycles in the factor graph, the problem is harder.

Discussion question:
What does it *mean* for a model to be a tree, compared to a non-tree model?