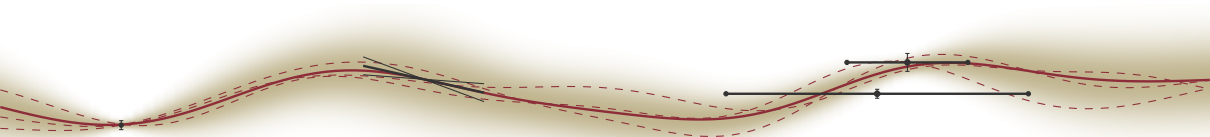
Probabilistic Inference and Learning
Lecture 17
The Sum-Product Algorithm

Philipp Hennig

17 December 2018

Eberhard Karls
Universität Tübingen

Faculty of Science
Department of Computer Science
Chair for the Methods of Machine Learning

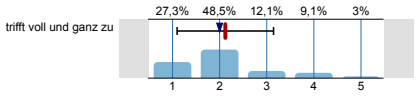


Feedback — Special Edition

Central Evaluation System

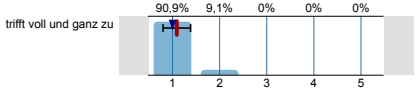
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

2.2) Der/die Dozent/in vermittelt die Sachverhalte verständlich.



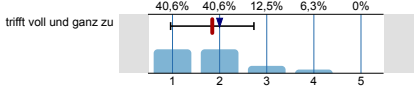
n=33
mw=2,1
md=2
s=1
E.=1

2.3) Der/die Dozent/in regt zur kritischen Auseinandersetzung mit den behandelten Themen an.



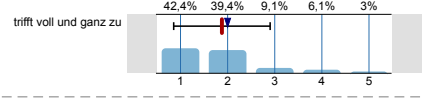
n=33
mw=1,1
md=1
s=0,3
E.=1

2.4) Der/die Dozent/in fördert die aktive Mitarbeit.



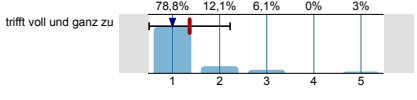
n=32
mw=1,8
md=2
s=0,9
E.=2

2.5) Die Arbeitsmaterialien (Folien, Skript, Handouts, ...) sind hilfreich.



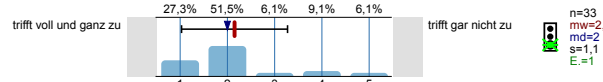
n=33
mw=1,9
md=2
s=1
E.=1

2.6) Der Besuch der Vorlesung lohnt sich.

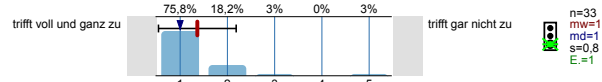


n=33
mw=1,4
md=1
s=0,9
E.=1

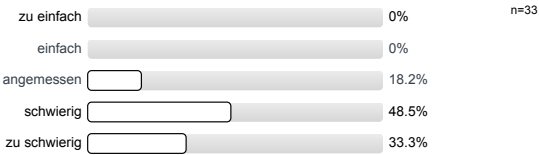
4.1) Die Leistungsanforderungen sind transparent.



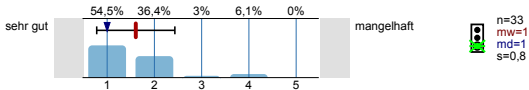
4.2) Die Veranstaltung fördert mein Interesse am Themengebiet.



4.3) Ich empfand die Veranstaltung als ...



4.4) Ich gebe der Veranstaltung die Gesamtnote:



Feedback — Special Edition

Free-Form Answers

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Key Problems

- Exercises too difficult
- Worries about Exam

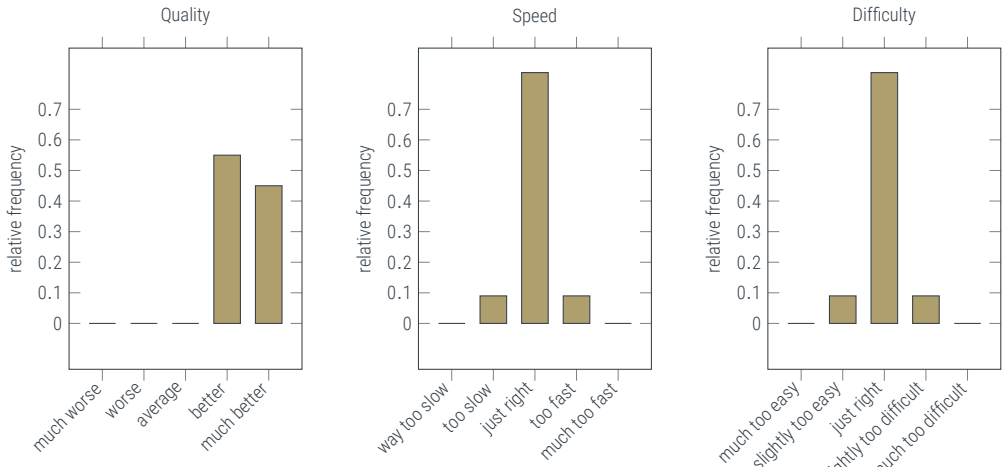
Key Strengths

- Lecture is motivating / stimulating, raises interest in the field
- you like responsiveness, and historical context

I will simplify the exercises, and make them more “exam-like”

Last Lecture: Debrief

Feedback dashboard



Last Lecture: Debrief

Detailed Feedback

Things you did not like:

- in ψ, the "p" is silent!

Things you did not understand:

- what is $\max_x p(x)$ in the context of graphical models?
- the recursive nature of message passing

Things you enjoyed:

- speed was *perfect* for the first time!
- connection to filters
- contexty of each framework
- outlook
- blackboard examples at the beginning

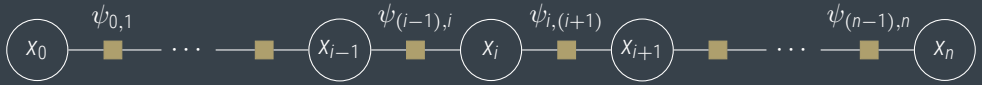
Overview of Lectures so far:

0. Introduction to Reasoning under Uncertainty
1. Probabilistic Reasoning
2. Probabilities over Continuous Variables
3. Gaussian Probability Distributions
4. Gaussian Parametric Regression
5. More on Parametric Regression
6. Gaussian Processes
7. More on Kernels & GPs
8. A practical GP example
9. Markov Chains, Time Series, Filtering
10. Classification
11. Empirical Example of Classification
12. Bayesianism and Frequentism
13. Stochastic Differential Equations
14. Exponential Families
15. Graphical Models
16. Factor Graphs
17. The Sum-Product Algorithm
18. Mixture Models

Today: Efficient Inference on Graphs

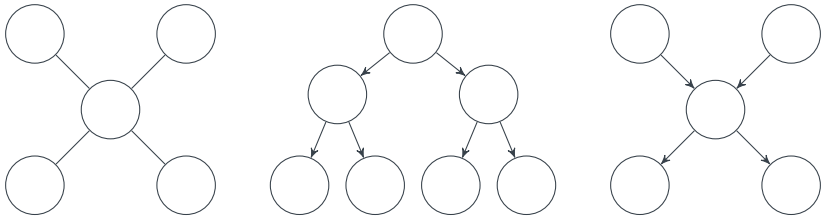
Factor Graphs

- are a tool to directly represent an entire computation in a formal language (which also includes the functions in question themselves)
- both directed and undirected graphical models can be mapped onto factor graphs.



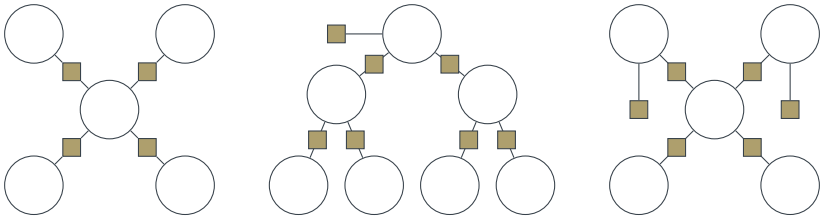
Inference on Chains

- separates into **local messages** being sent forwards and backwards along the factor graph
- both the local marginals and the *most-probable state* can be inferred in this way



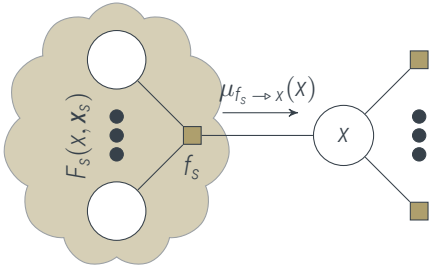
Definition (Tree)

An *undirected* graph is a **tree** if there is one, and only one, path between any pair of nodes (such graphs have no loops). A *directed* graph is a **tree** if there is only one node which has no parent (the *root*), and all other nodes have only one parent. When such graphs are transformed into undirected graphs by moralization, they remain a tree. A directed graph such that every pair of nodes is connected by one and only one path is called a **polytree**. When transformed into an undirected graph, such graphs, in general, acquire loops. But the corresponding factor graph is still a tree.



Definition (Tree)

An *undirected* graph is a **tree** if there is one, and only one, path between any pair of nodes (such graphs have no loops). A *directed* graph is a **tree** if there is only one node which has no parent (the *root*), and all other nodes have only one parent. When such graphs are transformed into undirected graphs by moralization, they remain a tree. A directed graph such that every pair of nodes is connected by one and only one path is called a **polytree**. When transformed into an undirected graph, such graphs, in general, acquire loops. But the corresponding factor graph is still a tree.



- Consider a tree-structured factor graph over $\boldsymbol{x} = [x_1, \ldots, x_n]$ (if instead you have an undirected tree or directed polytree, transform it first).
- Again, w.l.o.g. assume discrete variables for simplicity (for continuous, replace sums by integrals).
- Pick any variable $x \in \boldsymbol{x}$. We can write

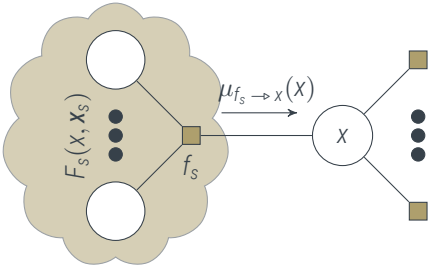
$$p(\boldsymbol{x}) = \prod_{s \in \mathrm{ne}(x)} F_s(x, \boldsymbol{x}_s)$$

where $\mathrm{ne}(x)$ are the **neighbors** of x, and F_s is the **sub-graph** of nodes $\boldsymbol{x}_s$ other than x itself that are connected to neighbor s (which is itself a tree!).

- Consider the **marginal** distribution $p(x) = \sum_{\boldsymbol{x} \setminus x} p(\boldsymbol{x})$

$$a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 = (a_1 + a_2) \cdot (b_1 + b_2)$$

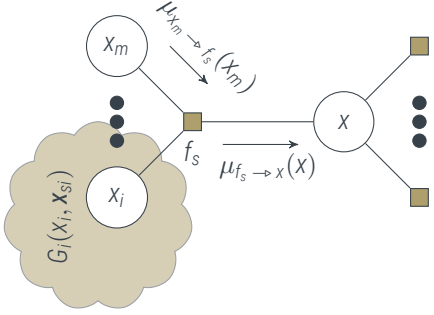
$$\sum_i \prod_j f_{ij} = \prod_j \sum_i f_{ij}$$



$$p(x) = \sum_{\boldsymbol{x} \setminus x} \prod_{s \in \mathrm{ne}(x)} F_s(x, \boldsymbol{x}_s) = \prod_{s \in \mathrm{ne}(x)} \underbrace{\left(\sum_{\boldsymbol{x}_s} F(x, \boldsymbol{x}_s) \right)}_{=: \mu_{f_s \to x}(x)}$$

$$= \prod_{s \in \mathrm{ne}(x)} \mu_{f_s \to x}(x)$$

The marginal $p(x)$ is a product of **incoming messages** $\mu_{f_s \to x}$ from the factors connected to x.



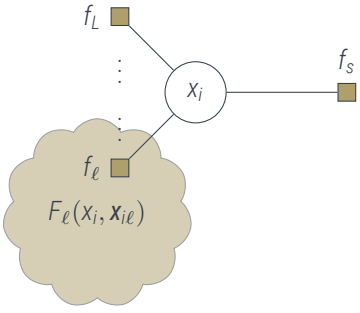
- consider the sub-graph $F_s(x, \mathbf{x}_s)$ and factorize **that** sub-graph into further (tree-structured) sub-graphs

$$F_s(x, \mathbf{x}_s) = f_s(x, x_1, \ldots, x_m) G_1(x_1, \mathbf{x}_{s1}) \cdots G_m(x_{1m}, \mathbf{x}_{sm})$$

- then we can write

$$\begin{aligned}\mu_{f_s \to x}(x) &= \sum_{x_1, \ldots, x_m} f_s(x, x_1, \ldots, x_m) \prod_{i \in \mathrm{ne}(f_s) \setminus x} \underbrace{\left(\sum_{\mathbf{x}_{si}} G_i(x_i, \mathbf{x}_{si}) \right)}_{\mu_{x_i \to f_s}(x_i)} \\ &= \sum_{x_1, \ldots, x_m} f_s(x, x_1, \ldots, x_m) \prod_{i \in \mathrm{ne}(f_s) \setminus x} \mu_{x_i \to f_s}(x_i)\end{aligned}$$

To compute the factor-to-variable message $\mu_{f_s \to x}(x)$, **sum** over the **product** of the factor and remaining sub-graph-sums. The latter are themselves **messages** from the variables connected to f_s.



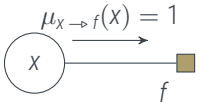
$$G_i(x_i, \boldsymbol{x}_{si}) = \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} F_\ell(x_i, \boldsymbol{x}_{i\ell})$$

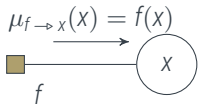
$$\mu_{x_i \to f_s(x_i)} = \sum_{\boldsymbol{x}_{si}} G_i(x_i, \boldsymbol{x}_{si}) = \sum_{\boldsymbol{x}_{si}} \left(\prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} F_\ell(x_i, \boldsymbol{x}_{i\ell}) \right)$$

$$= \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} \left(\sum_{\boldsymbol{x}_{i\ell}} F_\ell(x_i, \boldsymbol{x}_{i\ell}) \right)$$

$$= \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} \mu_{f_\ell \to x_i}(x_i)$$

To compute the variable-to-factor message $\mu_{x_i \to f_s}(x_i)$, take the **product** of all incoming factor-to-variable messages. Repeat recursively, until reaching a **leaf** node.





$$\mu_{x \to f}(x) = \prod_{\varnothing} \sum_{\varnothing} := 1$$

$$\mu_{f \to x}(x) = \sum_{\varnothing} f(x, \varnothing) \prod_{\varnothing} := f(x)$$

To initiate the messages at leaves of the graph, define them to be unit for variable leaves and identities for factor leaves.

The Sum-Product Algorithm

Summary

To compute the marginal $p(x)$, treat it as the root of the tree, and do:

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from leaves towards root x:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- at the root x, take product of all incoming messages (and normalize).

The Sum-Product Algorithm

Summary

To compute the marginals $p(x)$ of **all** variables, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_i} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**.
- once all nodes have received messages from all their neighbors, take product of all incoming messages at all variables (and normalize).

Inference on the marginal of all variables in a tree-structured factor-graph is **linear** in graph size.

- The two types of messages can be combined, phrasing the algorithm as message passing between factor nodes only:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

$$\mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

$$m_{f_\ell \to f_j}(\boldsymbol{x}_j) = \sum_{\boldsymbol{x}_\ell \setminus (\boldsymbol{x}_\ell \cap \boldsymbol{x}_j)} f_\ell(x_j, \boldsymbol{x}_\ell) \prod_{i \in \mathrm{ne}(f_\ell) \setminus \mathrm{ne}(f_j)} m_{f_i \to f_j}(x_\ell)$$

- There is a generalization from trees to general graphs, known as the **junction tree algorithm**. The principal idea is to **join** sets of variables in the graph into larger maximal cliques until the resulting graph is a tree. The exact process, however, requires care to ensure that every clique that is a sub-set of another clique ends up in that clique. The resulting algorithm (like the sum-product algorithm) has complexity exponential in the dimensionality of the largest variable in the graph, and linear in the size of tree.

The computational cost of probabilistic inference on the marginal of a variable in a joint distribution is exponential in the dimensionality of the maximal clique of the juntion tree, and linear in the size of the junction tree. The junction tree algorithm is **exact** for any graph (it produces correct martginals), and **efficient** in the sense that, given a graph, there does not in general (i.e. without using properties of the functions instead of the graph) exist a more efficient algorithm.

- If one or more nodes $\boldsymbol{x}^o$ in the graph are **observed** ($\boldsymbol{x}^o = \hat{\boldsymbol{x}}^o$), just introduce factors $f(x_i^o) = \delta(x_i^o - \hat{x}_i^o)$ into the graph.
- This amounts to "clamping" the variables to their observed value
- Say $\boldsymbol{x} := [\boldsymbol{x}^o, \boldsymbol{x}^h]$. Because $p(\boldsymbol{x}^o, \boldsymbol{x}^h) \propto p(\boldsymbol{x}^h \mid \boldsymbol{x}^o)$, the sum-product algorithm can thus be used to compute *posterior* marginal distributions over the hidden variables $\boldsymbol{x}^h$.

What if we don't care about the marginal posteriors,
but about the **joint** distribution?

In general, it's shape can be very complex, and exponentially hard to track (in the number of variables). But remember from lecture 1 that *storing* the **maximum** of the distribution has linear complexity (just write it down!).

How about **computing** that maximum?

The Max-Product / Max-Sum Algorithm

Finding Most Probable Configurations

[Exposition from Bishop, PRML, 2006]

- What if, instead of marginals $p(x_i)$ we want the jointly **most probable** state $x^{\max} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{x})$?
- note that $\arg\max_{\boldsymbol{x}} p(\boldsymbol{x}) \neq \prod \arg\max_{x_i} p(x_i)$:

		0.6	0.4
		$x_2 = 0$	$x_2 = 1$
0.7	$x_1 = 0$	0.3	**0.4**
0.3	$x_1 = 1$	0.3	0.0

- but $\max(ab, ac) = a \max(b, c)$ and $\max(a + b, a + c) = a + \max(b, c)$! Also (cf. earlier lectures)

$$\log\left(\max_{\boldsymbol{x}} p(\boldsymbol{x})\right) = \max_{\boldsymbol{x}} \log p(\boldsymbol{x})$$

Thus, we can compute the most probable state $x^{\max}$ by taking the sum-product algorithm and replacing all summations with maximizations (the **max-product** algorithm). We can further replace all products of p with sums of $\log p$ (the **max-sum** algorithm). The only complication is that, if we also want to know the arg max, we have to track it separately, using an additional data structure.

The Max-Product Algorithm

Summary

To compute $\boldsymbol{x}^{\max}$, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j}(x_j) = \max_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- additionally track indicator for **identity** of maximum (nb: This is a function of x_j!

$$\phi(x_j) = \arg\max_{x_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**. At each factor node, set $\boldsymbol{x}_{\ell j}^{\max} = \phi(x_j)$ (this is known as **backtracking**).

The Max-Sum Algorithm

Summary

To compute $\boldsymbol{x}^{\max}$, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = \log f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 0$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j}(x_j) = \max_{\boldsymbol{x}_{\ell j}} \log f_\ell(x_j, \boldsymbol{x}_{\ell j}) + \sum_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \sum_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- additionally track indicator for **identity** of maximum (nb: This is a function of x_j!)

$$\phi(x_j) = \arg\max_{x_{\ell j}} \log f_\ell(x_j, \boldsymbol{x}_{\ell j}) + \sum_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**. At each factor node, set $\boldsymbol{x}_{\ell j}^{\max} = \phi(x_j)$ (this is known as **backtracking**).

Assume discrete $x_i \in [1, \dots, k]$ for the moment. What is the **marginal** $p(x_i)$?

$$p(\boldsymbol{x}) = \frac{1}{Z}\psi_{0,1}(x_0, x_1) \cdots \psi_{i-1,i}(x_{i-1}, x_i) \cdot \psi_{i,i+1}(x_i, x_{i+1}) \cdot \psi_{n-1,n}(x_{n-1}, x_n)$$

$$p(x_i) = \sum_{x_{\neq i}} p(\boldsymbol{x}) = \underbrace{\frac{1}{Z}\left(\sum_{x_{i-1}} \psi_{i-1,i}(x_{i-1}, x_i) \cdots \left(\sum_{x_1} \psi_{1,2}(x_1, x_2)\left(\sum_{x_0} \psi(x_0, x_1)\right)\right)\right)}_{=:\mu_{\rightarrow}(x_i)}$$

$$\cdot \underbrace{\left(\sum_{x_{i+1}} \psi_{i,i+1}(x_i, x_{i+1}) \cdots \left(\sum_{x_n} \psi_{n-1,n}(x_{n-1}, x_n)\right)\right)}_{=:\mu_{\leftarrow}(x_i)} = \frac{1}{Z}\mu_{\rightarrow}(x_i) \cdot \mu_{\leftarrow}(x_i).$$

Base Case: Markov Chains

The Viterbi Algorithm

Assume discrete $x_i \in [1, \dots, k]$ for the moment. Where is the **maximum** $\max p(\mathbf{x})$?

$$p(\boldsymbol{x}) = \frac{1}{Z}\psi_{0,1}(x_0, x_1) \cdots \psi_{i-1,i}(x_{i-1}, x_i) \cdot \psi_{i,i+1}(x_i, x_{i+1}) \cdot \psi_{n-1,n}(x_{n-1}, x_n)$$

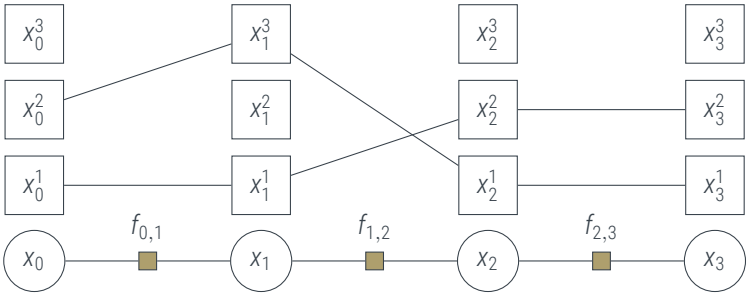
$$\begin{aligned}\max_{\boldsymbol{x}} p(\boldsymbol{x}) &= \frac{1}{Z} \max_{x_0} \cdots \max_{x_N} \psi_{0,1}(x_0, x_1) \cdots \psi_{n-1,n}(x_{n-1}, x_n) \\ &= \max_{x_0, x_1} \left(\psi_{0,1}(x_0, x_1) \left(\cdots \max_{x_n} \psi_{n-1,n}(x_{n-1}, x_n) \right) \right)\end{aligned}$$

$$\arg\max_{\boldsymbol{x}} p(\boldsymbol{x}) = \arg\max_{x_0, x_1} \left(\log \psi_{0,1}(x_0, x_1) + \left(\cdots + \arg\max_{x_n} \log \psi_{n-1,n}(x_{n-1}, x_n) \right) \right)$$

The Viterbi Algorith

On a **trellis diagram**

[based on Fig.8.53 in Bishop, PRML, 2006]



$$\mu_{x_0 \to f_{01}} = 0$$

$$\mu_{f_{i-1,i} \to x_i}(x_i) = \max_{x_{i-1}} \left(\log f_{i-1,i}(x_{i-1}, x_i) + \mu_{x_{i-1} \to f_{i-1,i}}(x_{i-1}) \right)$$

$$\phi(x_i) = \arg\max_{x_{i-1}} \left(\log f_{i-1,i}(x_{i-1}, x_i) + \mu_{x_{i-1} \to f_{i-1,i}}(x_i) \right)$$

$$\mu_{x_i \to f_{i,i+1}}(x_i) = \mu_{f_{i-1,i} \to x_i}(x_i)$$

$$x_{i-1}^{\max} = \phi(x_i^{\max})$$

- Max-Sum is a case of **dynamic programming** (recursive simplification of optimization using problem structure). The equation

$$\mu_{f_\ell \to f_j} = \operatorname*{arg\,max}_{\boldsymbol{x}_\ell \setminus (\boldsymbol{x}_\ell \cap \boldsymbol{x}_j)} \left(\log f_\ell(x_\ell) + \sum_{i \in \mathrm{ne}(f_\ell) \setminus \mathrm{ne}(f_j)} \mu_{f_i \to f_j}(\boldsymbol{x}_j) \right)$$

  defines a **Hamilton-Jacobi-Bellman equation**

Summary:

- Factor graphs provide graphical representation of joint probability distributions that is particularly conducive to automated inference
- In factor graphs that are *trees*, all **marginals** can be computed in time **linear** in the graph size by **passing messages** along the edges of the graph using the **sum-product** algorithm.
- Computation of each local marginal is exponential in the dimensionality of the node. Thus, in general, the cost of inference is exponential in clique-size, linear in clique-number.
- An analogous algorithm, the **max-sum** algorithm, can be used to find the joint most probable state, also in linear time.
- Both algorithms fundamentally rest on the distributive properties

$$a(b+c) = ab + ac \qquad \max(ab, ac) = a \cdot \max(b, c)$$

Message passing provides the general framework for managing computational complexity in probabilistic generative models as far as it is caused by **conditional independence**. It does not, however, address complexity arising from the algebraic form of continous probability distributions. We already saw that **exponential families** address this latter issue. But not every distribution is an exponential family. A main theme for the remainder will be how to project complicated joint distributions onto factor graphs of exponential families.