Proposed design for gR, a graphical models toolkit for R

Kevin P. Murphy
www.ai.mit.edu/∼murphyk

29 September 2003

Contents

1 Introduction

This document is an extension of the talk I gave at the gR meeting[1] in Aalborg on 19 September 2003. It outlines a proposed design for gR, a graphical models library for R[2]. This design is similar to the design of BNT[3], but is much more general, in that it supports undirected models and chain graphs, and allows parameters to be represented as random variables (Bayesian modeling).

1.1 Comparison of proposed design with existing software

There are a large number of software packages for graphical models in current use[4], but none come close to the level of generality, flexibility and openness that this design aspires to. Here I briefly mention a few of the closest "competitors":

- BNT probably comes closest, and implements about 60% of the functionality described here. The main functionality it lacks is that BNT does not support Bayesian modeling, and only has limited support for undirected models and no support for chain graphs.
- PNL[5] currently implements about half the functionality of BNT (see discussion in Section 6). The main things it lacks are inference for Gaussian models, decision networks, and good support for DBNs and structure learning.

[1] http://www.r-project.org/gR
[2] R is essentially an open-source version of S-plus. See www.r-project.org
[3] Bayes Net Toolbox for Matlab. See www.ai.mit.edu/∼murphyk/Software/BNT/bnt.html
[4] See www.ai.mit.edu/∼murphyk/Software/BNT/bnsoft.html for a comparison of existing software for graphical models.
[5] Probabilistic Networks Library (Intel's C++ version of BNT). See www.intel.com/research/mrl/pnl/

- BUGS[6] implements about 30% of the functionality described here. It handles Bayesian models, but not decision networks, DBNs or structure learning. Also, it only has one kind of inference engine (Gibbs sampling). and is not open source.
- GMTk[7] is designed for DBNs for speech recognition. It supports parameter estimation and structure learning, but only has one kind of inference engine (junction tree), and does not support general graphical models. Also, it is not currently open source.
- OpenBayes[8] was started by Richard Dybowski. This project never took off the ground, despite much discussion.
- "Software Support for Bayesian Analysis Systems". Various systems were described at this workshop held at NIPS 2000[9] and at Interface 2001[10]. However, none are publicly available (except BNT).
- AutoBayes[11] [FS03, GFSB02]. The AutoBayes system uses theorem proving techniques to find symbolic solutions to certain graphical model inference/learning problems. For example, it can automatically reproduce textbook examples of Bayesian analysis of conjugate-exponential models in the fully observed data regime. Unfortunately, small changes to the model (e.g., non conjugate priors or latent variables/missing data) often render it intractable to solve analytically. In such a case, it generates model-specific code to compute the answer using numerical integration. (Clearly other approximations could be used.) It can also generate EM-based algorithms. Although the system is in principle very powerful, in practice the ability of a general purpose theorem prover to recognize which algorithm/schema to use is limited. Also, the system is not open source.

1.2 Graphical model communities

Graphical model technology is currently being developed in several different communities, each with their own terminology and philosophical outlook. Ideally gR will be catholic enough in its approach to be of interest to researchers in each of these communities. More importantly, gR should be useful to users of GM technology, who span an even wider range of communities, such as genetics, forensic science, epidemiology, bio-informatics, fault diagnosis, geostatistics, machine learning, data mining, computer vision, text processing, speech recognition, robotics, error-correcting codes, etc.

I come from the UAI/NIPS/Matlab community, which is reflected in the design outlined in this document. The design will no doubt need modifying to satisfy the needs/style of the statistics/R community. However, hopefully it will prove possible to "tunnel through" (to quote Steffen Lauritzen) from the machine learning community to the stats community (and vice versa!).

1.3 Summary of proposed design

At the highest level of abstraction, we can identify the following major topics:

- Representation - how do we specify the graph structure and the parametric forms of the local distributions?
- Non-parametric reasoning - computations on the graph structure
- Inference - state estimation
- Learning - parameter estimation and model selection
- Control - decision making

For representation, we need the following main classes:

- Graph

[6]Bayesian inference Using Gibbs Sampling. See http://www.mrc-bsu.cam.ac.uk/bugs/.
[7]Graphical Models Toolkit. See ssli.ee.washington.edu/∼bilmes/gmtk/
[8]See www.ai.mit.edu/∼murphyk/OpenBayes/index.html
[9]http://ase.arc.nasa.gov/nips2000/
[10]http://www.ics.uci.edu/∼interfac/all-sessions.html#FBI5
[11]See http://ase.arc.nasa.gov/docs/autobayes.html

- LocalFactor - this can be a CPD (conditional probability distribution), which sums to one, or a Potential, which is an arbitrary positive function of its arguments.[12]
- GraphicalModel - this is just a graph plus a list of local factors. We will also consider various special-purpose graphical model classes.

For non-parametric reasoning, we need to provide a set of graph functions e.g., to assess (causal) identifiability [Pea00].

Once we have specified the parameters of the model, we can do many more things with it, such as:

- Infer hidden variables given a single case of evidence.
- Estimate (learn) parameters given a set of data cases (model fitting).[13]
- Sample values (if it's a generative, as opposed to discriminative, model).
- Make optimal decisions (if it's a decision network).

Although these may be thought of as methods of the GM class, in fact there are many algorithms to perform each of these tasks. Since these algorithms often have internal state (e.g., auxiliary data structures such as a junction tree, or storing internal parameters, such as the number of samples to collect), it is better to think of these algorithms as objects, which take the GM (which is just a passive data structure) as an argument. Hence we will define inference and learning "engines" below. Learning must call inference as a subroutine (except in the important special case of fully observed DAG models).

This suggests the following extra classes

- InfEngine - inference engine
- ParamLearnEngine - for ML/MAP parameter estimation
- StructLearnEngine - structure learning
- DecisionEngine - for decision networks (influence diagrams)

We will discuss all of these in detail below.

2 Representation

2.1 Graphs

2.1.1 Class or structure?

The graph class should presumably be compatible with the R graph package[14], although this does not seem to support mixed directed-undirected graphs. Also, I think it is better to represent graphs using a "lightweight" structure, such as a C-style struct, rather than a "heavy" class with lots of unrelated methods, which the current R graph class suffers from. Using a struct rather than a class means that functions (unlike methods) can be added independently by different developers (see discussion in Section 6), and can be grouped/stored according to functionality, rather than forcing them all to be centrally stored in the graph class. (Imagine adding all the functions in Figure 12 to the graph class definition! Why should someone who never uses the junction tree algorithm have to see these functions?)

The graph should just represent the topology. Extra information about the nodes and arcs is usually specific to a particular parameterization of a graph, and is better kept inside the GM object than inside the graph object. Functions that need to know about node/edge types can just take extra arguments (we will see examples below).

[12] "LocalFactor" is perhaps not a good name, since the term "factor" has another meaning in R. We need a word that describes the union of potentials and CPDs. "LocalDistribution" is possible, except a potential is not (necessarily) a distribution. The term "factor" was introduced in [KFL01] to mean a term in the product defining the joint. "Local factor" is a compromise, to disambiguate from the R usage.

[13] Bayesians do not distinguish between parameters and other latent variables, and hence for them, parameter learning (estimation) is just a special case of inference.

[14] Available at www.bioconductor.org

2.1.2 Kinds of graphs

It will be useful to distinguish different kinds of graphs, such as DAGs, undirected graphs, mixed directed-undirected, bipartite, etc. These could be defined as subclasses, but this seems quite “heavy”; I prefer to minimize inheritance and subtyping. Instead, it will probably suffice to add a ’type’ field to the struct, which specifies what kind of graph this is.

2.1.3 GUIs

There has been some discussion of GUIs for R[15], but none of this pertains to graphs or graphical models. In the UAI community, the GeNIe[16] package is considered one of the best GUIs, but it is only freely-available in binary form, and is restricted to DAG models.

There are many things the GUI should be able to support, such as:

- Draw a graph. (We want to allow for different types of nodes and edges).
- Specify the local factors interactively. (Each local factor class should provide a method for creating/updating itself via a GUI.)
- Specify queries (e.g., click on desired nodes).
- Specify evidence/ training data (e.g., from a file).
- Call inference and learning engines. (Each engine should provide a method for calling itself via a GUI.)
- Display results of query e.g., as histogram, Gaussian 2D ellipsoid, quantile plots, etc. (Each JPD should provide a method for displaying itself, presumably using built-in R functions.)
- Display results of structure learning (see Section 2.1.4).
- Call arbitrary R functions.

This is clearly a large amount of work, and is probably not appropriate for the gR project to attempt to do all this (in my opinion).

2.1.4 Graph visualization

The graph class should call Graphviz[17] to do automatic graph layout using the Rgraphiz package[18]. This is useful for visualizing the results of structure learning. The system should be able to represent the confidence that an arc is present. See, for example, Nir Friedman’s pathway explorer for visualizing features of models learned from biological data[19]. The confidence that each arc is present is indicated by shading; nodes can be clicked on, to pull up relevant pubMed citations/data, etc.

The GUI and the graph visualizer should be integrated, so one can create an initial model, fit it to data (modifying its structure), visualize the resulting fitted model, edit the result (by manually changing some arcs), and repeating the cycle.

2.1.5 File formats

Much time has been spent in the UAI community discussing file formats for Bayes nets[20]. The emphasis is on how to represent the CPDs, rather than representing the graph (which is comparitively easy). However, all of the current proposals are much too restrictive for our purposes, since they cannot handle continuous variables, Bayesian models, undirected models, DBNs, etc. gR should probably just define its own XML-based format, and then provide converters for other popular formats such as Hugin and BUGS if demand warrants.

[15]See http://www.sciviews.org/_rgui/

[16]www.sis.pitt.edu/∼genie/

[17]www.research.att.com/sw/tools/graphviz/

[18]Available at www.bioconductor.org

[19]http://www.cs.huji.ac.il/labs/compbio/ismb01/Rose/index1.html

[20]See e.g., www.ai.mit.edu/∼murphyk/OpenBayes/file_formats.html.

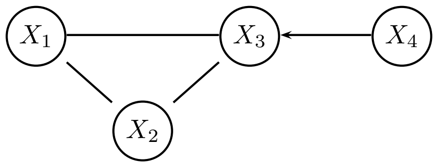


Figure 1: A chain graph model

2.1.6 Creating the graph via an API

The best way to create complex models is via a scripting language. I will illustrate this in Section 2.3 using a Matlab-like syntax (it should be easy to convert the examples to R). This is to emphasize that we are not proposing to write a parser to intrepret the model specification language (as in BUGS); instead, the models are created by calling functions in the API. This allows us to mix computation and model specification.

A graph can be specified by creating an adjacency matrix. But we need a way to refer to the nodes. I shall use the syntax

$$[X(1:n), Y(1:n)] = 1:N$$

to mean that X(1)=1, X(2)=2, ..., Y(1)=n+1, ..., Y(n)=N. This allows us to associate variable names with node numbers. For example, consider the toy chain graph in Figure 1. We can create this graph as follows:

```
X(1:4) = 1:4;
G = zeros(4,4);
G(X(1:3), X(1:3)) = 1;
G(X(4), X(3)) = 1;
```

We will see more complex examples later on.

2.1.7 Accessing the graph

There are some trivial accessor functions which we need, which, amazingly, are not built in to the standard R graph class:

- `m = markovBlanket(G, i)`
- `nbrs = neighbors(G: UndirGraph, i)`
- `ps = parents(G: DAG, i)`
- `cs = children(G: DAG, i)`
- `e = edgeNum(G, i, j)`
- `c = clqNum(G, node list)`

The clqNum function returns a unique identifier for each clique in the graph (the argument is the set of node numbers within the clique). Hence we can talk of the i'th clique, etc. Similarly for edges (pairwise cliques).

2.1.8 Graph algorithms

There are a large number of functions we need to perform on graphs. Rather than making them all methods of the graph class, I suggest that they be grouped according to functionality, and packaged separately. Some obvious groups include:

- Functions related to making junction trees (see Figure 12).
- Functions related to structure learning (see Section 4.2).

- Functions related to non-parametric reasoning with graphical models (see e.g., discussion of identifiability in [Pea00]).
- "Generic"/ "standard" graph-theoretic algorithms, such as mininum spanning tree, depth first search, topological sorting, etc. These can be added on an as-needed basis.

2.2 Local factors

The local factors define the parameterization of the model. A local factor can be a CPD (conditional probability distribution), which sums to one, or a Potential, which is an arbitrary positive function of its arguments. We will see examples of both below.

We use the following notation: $P(Y|X;\theta)$ is a CPD for child Y given parents X with fixed parameters θ; in a Bayesian setting, we write this as $P(Y|X,\theta;\alpha)$, where the parameters θ are random variables, and α are the fixed hyper-parameters. (Obviously we can repeat this recursively.) For undirected models, we write $\Psi(X,Y;\theta)$ where Ψ is a potential functions that may not sum to one. For Bayesian undirected models, we write $\Psi(X,Y|\theta;\alpha)$. We discuss chain graphs below.

We now summarize the proposed local factor classes, and give examples of their use in Section 2.3.

2.2.1 CPDs

- TabularCPD
- SparseTabularCPD - only a small number of non-zero entries
- SoftmaxCPD - binary case is called sigmoid [Nea92]
- NoisyOrCPD [Pea88] and its generalizations
- ClassificationTreeCPD [BFGK96]
- RegressionTreeCPD
- GLMCPD - Generalized Linear Model
- MLPCPD - Multi Layer Perceptron (feedforward neural network)
- DeterministicCPD
- CondLinGaussCPD - conditional linear Gaussian (see below)
- MixGaussCPD - mixture of Gaussians
- DirichletCPD - prior for TabularCPD
- GammaCPD - prior for variance (scalar)
- WishartCPD - prior for covariance matrix
- GaussianCPD - prior for vectors
- MatrixGaussianCPD - prior for weight matrices [Daw81, Min00]

For continuous children, we will often want to use a conditional linear Gaussian (CLG) distribution. For example, consider the network fragment $X \rightarrow Y \leftarrow C$. We say Y has a CLG distribution if

$$P(Y|X,C=i,\mu,\Sigma,W) = \mathcal{N}(Y;W_iX+\mu_i,\Sigma_i)$$

Special cases include: linear Gaussian (no conditioning on C), conditional Gaussian (no regression on X), and unconditional Gaussian (neither C or X exist).

If there is more than one discrete parent, then i indexes their joint assignment. Since this increases the number of parameters exponentially, there are two solutions. First, If μ_i is independent of i, we say the mean is tied across states of the conditioning variable(s); similarly for Σ and W. Second, we can regard some of the discrete parents as regression variables (representing their values using a 1-of-n binary encoding), instead of as indices. This should be an optional argument to the CLG/GLM constructor.

Some CPDs may also allow some parents to act as “switches”, as in the GMTk package[21]. This means that, conditional on the value of the switch, only a subset of the other parents may be relevant, as in a multiplexer (this is a way of encoding context-specific independence [BFGK96]).

It is tempting to go hog-wild and throw in all kinds of probabilistic classification and regression methods here. However, each CPD class must be capable of implementing the methods discussed in Section 2.2.4.

2.2.2 Potentials

- TabularPot
- SparseTabularPot
- GaussianPot
- MixGaussPot
- LogLinPot - log linear potential: see Section 2.3.3

2.2.3 Decision networks

- TabularDecisionNode
- TabularUtilityNode
- GaussianDecisionNode
- GaussianUtilityNode

Decision networks are discussed in Section 2.3.5.

2.2.4 Methods which local factors must implement

The methods that each local factor class must implement fall into two main camps: methods for infernece and methods for parameter learning. We discuss these in Sections 3.4 and 4.1 respectively.

2.3 Graphical Model class

The list of local factors completely defines the graphical model, since we can infer the graph from their domains. We will give examples of this below. Hence we can create a graphical model directly from the list of factors:

```
GM = mkGM(list of local factors);
```

This assumes that the graph structure is static/ fixed-sized. If we have variable-sized data (e.g., variable-length sequences), we need to specify how the system should tie parameters as the network gets “unrolled” over time: see Section 2.3.10.

When the user creates a GM by passing in the list of local factors, the constructor can examine the model and compute certain properties (which can be stored as member fields inside the GM), such as

- Are there any decision/utility nodes?
- Which nodes are discrete, and which continuous?

[21]ssli.ee.washington.edu/∼bilmes/gmtk/

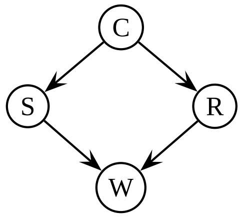


Figure 2: A simple Bayes net. C=cloudy, S=Sprinkler, R=rain, W=wet grass. The wet grass can either be caused by rain or by a water sprinkler. Clouds make it less likely the sprinkler will turn on, but more likely it will rain.

- What sizes do the nodes have? (For discrete nodes, the size is the number of possible values; for continuous, is the length of the vector.)
- Is the graph a DAG (Bayes net), a directed cyclic graph (dependency network [HCM$^+$00]), an undirected graph (Markov net), or is it a DAG on blocks (chain graph)?
- Are all clique potentials defined on at most two nodes (pairwise/ edge based potentials)?
- Is the graph a tree or does it have loops?
- Are all the parameters constant (frequentist) or are some random variables (Bayesian)?
- Are all the latent nodes discrete? (This makes inference much easier.) To answer this, the user has to specify (as an optional argument to the GM constructor) which nodes are latent; all the rest are assumed to be observed.

On the basis of these properties, the constructor can figure out if the model falls into any of the traditional model classes. If all the potentials are CPDs, and the graph is a DAG, then the GM is in fact a Bayes net. This class is special since ML/MAP parameter estimation does not require inference (assuming fully observed data); this follows from the fact that the CPDs are locally normalized. (This fact partly explains the popularity of BNs.) Hence it is worth automatically detecting if the GM happens to be a BN. But it will usually be easier for the user to specify explicitly that he/she wants to create a specific kind of model. These special purpose models can have specialized algorithms for inference and learning. This provides a wrapper mechanism for using existing software.

Note: As with graphs, I propose to treat graphical models as a "lightweight" data structure, rather than an object that is capable of performing operations. Heavy computation will be performed by the "engine" classes. The only methods that the GM class should need implement are accessor functions, to get/set the graph and/or local factors, etc (or we could simply make these public fields of a struct).

2.3.1 DAGs

Consider the water sprinkler network (from [RN02]) in Figure 2.

The joint is

$$P(C,R,S,W) = P(C;\theta_C)P(R|C;\theta_R)P(S|C;\theta_S)P(W|R,S;\theta_W)$$

This model is easy to create:

```
C = 1; R = 2; S = 3; W = 4;
CPD{C} = TabularCPD(table = thetaC, child = C)
CPD{R} = TabularCPD(table = thetaR, child = R, parents = C)
CPD{S} = TabularCPD(table = thetaS, child = S, parents = C)
CPD{W} = TabularCPD(table = thetaW, child = W, parents = [R S])
```

where thetaC is a 1D array (which sums to one), thetaR and thetaS are 2D arrays (where each row sums to one), and thetaW is a 3D array (with the appropriate sum-to-one constraint). The size of these arrays implicitly defines the size of the variables, i.e., the number of discrete values (levels) these variables can take on. Note that, unlike in BNT, these CPDs are "free floating" objects, not yet associated with a particular graphical model.

Now consider the Rats example in Figure 3, which illustrates Bayesian modeling with plates. We can create this as follows.

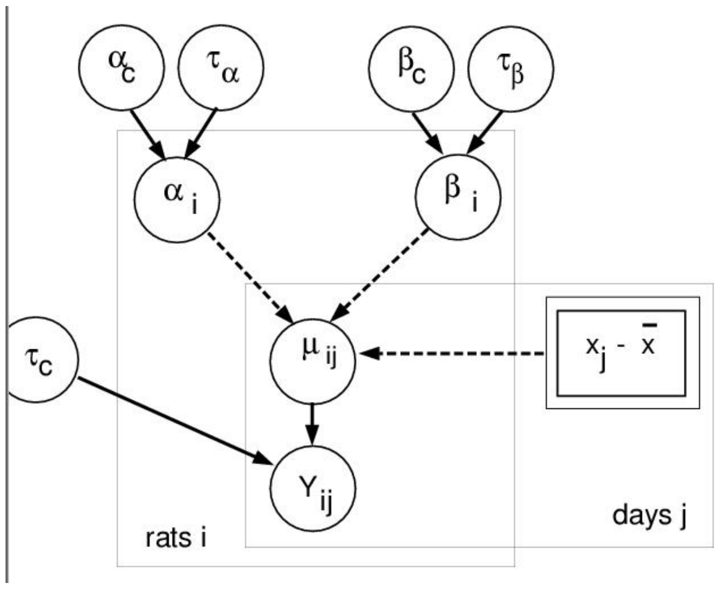


Figure 3: The Rats model using plate notation, from the BUGS example manual (volume 1).

```
x.bar = mean(x);
Nrats = 30;
Ndays = 5;
Nnodes = 5 + 2*Nrats + 2*Nrats*Ndays;
[Xtau.x, Xalpha.c, Xalpha.tau, Xbeta.c, Xbeta.tau, Xalpha(1:Nrats),
  Xbeta(1:Nrats), Xmu(1:Nrats,1:Ndays), Y(1:Nrats,1:Ndays)] = 1:Nnodes;
CPD{Xalpha.c} = gaussianCPD(mean = 0, precision = 1e-4, child=Xalpha.c);
CPD{Xbeta.c} = gaussianCPD(mean = 0, precision = 1e-4, child=Xbeta.c);
CPD{Xtau.c} = gammaCPD(param1 = 1e-3, param2 = 1e-3, child=Xtau.c);
CPD{Xtau.alpha} = gammaCPD(param1 = 1e-3, param2 = 1e-3, child=Xtau.alpha);
CPD{Xtau.beta} = gammaCPD(param1 = 1e-3, param2 = 1e-3, child=Xtau.beta);
for i=1:Nrats
  CPD{Xalpha(i)} = gaussianCPD(child = Xalpha(i),
                    meanParent = Xalpha.c, precisionParent =  Xtau.alpha);
  CPD{Xbeta(i)} = gaussianCPD(child=Xbeta(i),
                    meanParent = Xbeta.c, precisionParent =  Xtau.beta);
  for j=1:Ndays
     CPD{Xmu(i,j)} = deterministicCPD(child = Xmu(i,j),
                               parents = [Xalpha(i), Xbeta(i)],
                               fn = lambda(a,b). a+b*(x(j)-x.bar));
     CPD{Y(i,j)} = gaussianCPD(child = Y(i,j),
                    meanParent = Xmu(i,j), precisionParent = Xtau.c);
  end
end
```

The plates get replaced by for loops, which are arguably easier to understand. Efficiency is not a concern here, since we are just specifying the model, and not doing any real computation.

Note that the deterministic CPD takes as argument a function, which is defined inline as an anonymous (lambda) function.

For vector-valued nodes, we would replace the gamma priors with Wisharts, and the normal priors with matrix-normals [Daw81, Min00].

Note that it should be possible to solve this model in closed form, since everything is in the conjugate-exponential

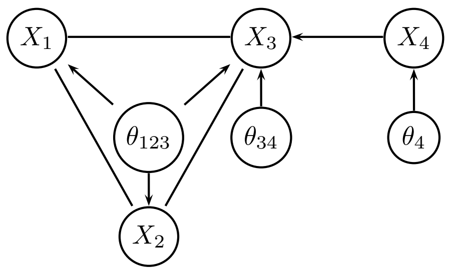


Figure 4: A Bayesian version of the chain graph model in Figure 1.

family and we have no missing data. (The deterministic node can be "compiled out" during a preprocessing step.) If there is missing data, we can use Gibbs sampling, variational methods or expectation propagation. We discuss inference in Section 3.

2.3.2 Chain graph

Consider the chain graph in Figure 1.

The joint is

$$P(X1:X4) = \frac{1}{Z(X4)}\Psi(X1,X2,X3;\theta_{123})\Psi(X3|X4;\theta_{34}) \times P(X4;\theta_4)$$

We can create this using

```
pot{clqNum(G,X(1:3))} = TabularPot(table = theta123, domain = X(1:3));
pot{clqNum(G,X(3:4))} = TabularPot(table = theta34, domain = X(3:4), parents = X(4));
pot{X(4)} = TabularCPD(table = theta4, domain = X(4));
```

where theta123 is a positive 3D array, and theta34 is a positive 2D array. (The size of the arrays implicitly defines the size of the state-spaces of the variables.) Note that it is helpful to define the graph before the potentials, so we can associate cliques with numbers.

We can create a Bayesian version of this as shown in Figure 4. The joint is

$$P(X_{1:4},\theta_{123},\theta_{34},\theta_4) = \frac{1}{Z(X_4,\theta_{123},\theta_{34})}\Psi(X_1,X_2,X_3|\theta_{123})\Psi(X_3|X_4,\theta_{34})\times P(X_4|\theta_4)\times P(\theta_{123};\alpha_{123})P(\theta_{34};\alpha_{34})P(\theta_4;\alpha_4)$$

We can create this as follows.

```
pot{clqNum(G,X(1:3))} = TabularPot(tableParent = Xtheta123,
    domain = [X(1:3) Xtheta123]));
pot{clqNum(G,X(3:4))} = TabularPot(tableParent = Xtheta34,
    domain = [X(3:4) Xtheta34], parents = X(4));
pot{X(4)} = TabularCPD(tableParent = Xtheta4, child = X(4));
pot{Xtheta123} = DirichletCPD(table = alpha123, child = Xtheta123);
pot{Xtheta34} = DirichletCPD(table = alpha34, child = Xtheta34);
pot{Xtheta4} = DirichletCPD(table = alpha4, child = Xtheta4);
```

Now consider the modification of the previous example shown in Figure 5, where X_4 is now a parent of all the nodes in the clique. The joint is

$$P(X1:X4) = \frac{1}{Z(X4)}\Psi(X1,X2,X3|X4;\theta_{1234}) \times P(X4;\theta_4)$$

We can create this as follows:

```
pot{clqNum(G,X(1:4))} = TabularPot(table = theta1234, domain = X(1:4), parents=X(4));
pot{X(4)} = TabularCPD(table = theta4, domain = X(4));
```

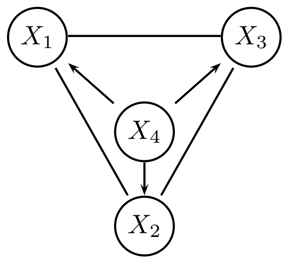


Figure 5: Another chain graph model, where X_4 is now a parent of all the variables in the clique.

2.3.3 Log-linear model

Consider a fully undirected version of Figure 1, but represent the potentials as log-linear functions:

$$P(X1:X4) = (1/Z) * \exp[\lambda_1' * f1(X1, X2, X3) + \lambda_2' * f2(X3, X4)]$$

Here, $f1(X1, X2, X3)$ is a feature vector for each assignment to $X1, X2, X3$. Suppose f1 returns the binary-encoding of its arguments, so f1(X1,X2,X3) is an 8-vector; then lambda1 is an 8-vector, with one free parameter for eacn entry in f1, so $\lambda_1 = \log \theta_{123}$, where θ_{123} is the fully parameterized clique potential.

However, f1 can be an arbitrary function, and may not produce a unique result for every assignment, thus requiring fewer parameters (smaller lambda vector). For example, suppose f1 is the OR function; then there are just two free parameters (so lambda1 is a 2-vector). We can specify this model as follows.

```
pot{clqNum(G,X(1:3))} = LogLinPot(param = lambda1, feature = f1, domain=X(1:3));
pot{clqNum(G,X(3:4))} = LogLinPot(param = lambda2, feature = f2, domain=X(3:4));
```

Things get more interesting when we don't use maximal cliques. For example, we may just use pairwise cliques (edges):

$$P(X1:X4) = (1/Z) * \exp[\lambda_1' * f1(X1, X2) + \lambda_2' * f2(X1, X3) + \lambda_3' * f3(X2, X3) + \lambda_4' * f4(X3, X4)]$$

The code becomes

```
for i=1:4
  for j=i+1:4
    if G(i,j)==1
       e = edgeNum(G, i, j);
       pot{e} = LogLinPot(param = lambda{e}, feature= f{e}, domain=[X(i) X(j)]);
```

Note that this model has conditional independencies that cannot be represented in a standard graphical model, but which could be represented in a factor graph [Fre03] (this is the undirected analog of local structure within a CPD).

2.3.4 Factor graphs

Factor graphs [KFL01, Fre03] provide a convenient representation that unifies directed and undirected graphical models. In a factor graph, there are two kinds of nodes, representing factors (local terms) and variables. Variable node X_i is connected to all factor nodes F_i which contain X_i in their domain. Hence factor graphs are bipartite. Variable nodes are represented by circles, and factor nodes by squares. See Figure 6 for an example. This specifies

$$P(X_{1:5}) \propto \Psi_{1,2}(X_1, X_2)\Psi_{2,3}(X_2, X_3)\Psi_{1,3}(X_1, X_3)\Psi_{3,4}(X_3, X_4)\Psi_{3,5}(X_3, X_5)$$

Note that the MRF representation cannot graphically represent the factorized form that we have assumed for the potential on clique X_1, X_2, X_3

$$\psi(X_1, X_2, X_3) = \psi(X_1, X_2)\psi(X_2, X_3)\psi(X_1, X_3).$$

We specify this model as follows.

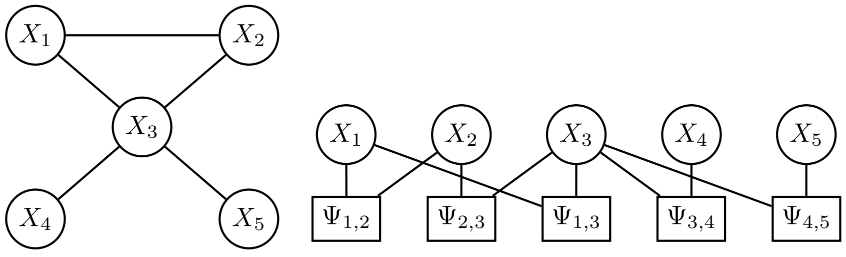


Figure 6: Converting the MRF in (a) to the factor graph in (b), where we have assumed $\psi(X_1, X_2, X_3) = \psi(X_1, X_2)\psi(X_2, X_3)\psi(X_1, X_3)$.

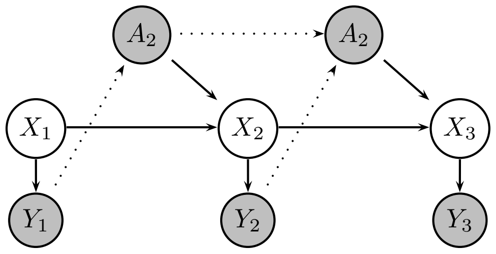


Figure 7: A POMDP where the information arcs (dotted) only come from the previous time-slice instead of the previous history. X_t is the hidden state, Y_t is the observation and A_t is the action. The reward nodes are not shown for simplicity, but have the same connectivity as the observation nodes, Y_t.

```
pot{edgeNum(G,X(1),X(2))} = TabularPot(domain=X(1:2), table = Psi12);
pot{edgeNum(G,X(2),X(3))} = TabularPot(domain=X(2:3), table = Psi23);
...
```

2.3.5 Decision networks

Consider the 3 time-slice POMDP (partially observed Markov decision process [KLC98]) in Figure 7. Let us suppose all variables are discrete.

The optimal model has information arcs from all past observations/actions/rewards, $y/a/r_{1:t-1}$, to the current decision node, A_t. But this is computationally intractable. Instead, consider a LIMID (limited memory influence diagrams [LN01]), where we only have information arcs from the previos time-slice (as in Figure 7). We can construct this as follows. (We hard-code the parameters, even though they are tied.)

```
for t=1:3
  CPD{Y(t)} = TabularCPD(child = Y(t), parents = X(t), table = thetaY);
  CPD{R(t)} = TabularUtilityNode(parents = [X(t) A(t)], table = thetaR);
end
CPD{X(1)} = TabularCPD(table = theta1);
for t=2:3
  CPD{X(t)} = TabularCPD(child=X(t), parents = [X(t-1) A(t)], table = thetaX);
  CPD{A(t)} = TabularDecisionNode(infoParents = [Y(t-1) R(t-1)], table = policy);
end
```

Note: this is similar to the way LIMIDs are specified in BNT.
Gaussian decision networks are discussed in [SK89].

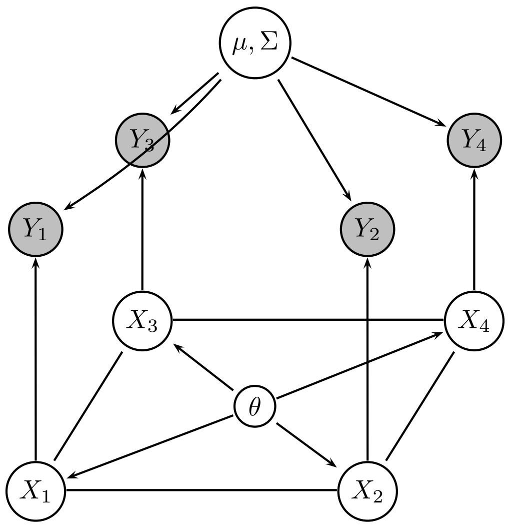


Figure 8: A Markov Random Field defined on a 2D lattice, with the parameters shown explicitly. Each hidden variable X_i has its own "private" observed child Y_i.

2.3.6 Frequentist models with tied parameters

Although Bayesian models are elegant, it is often very hard to do inference in them. A common alternative is to clamp the parameters to a point value (usually the maximum likelihood/ MAP estimate) and then remove them from the graph. The resulting graphical model is often more amenable to exact inference.

To find the ML/MAP parameter estimates, we must assume the parameters are tied (shared) across data cases, so the graph structure looks identical to the Bayesian case.[22] We may even put priors on the parameters (for MAP estimation). The only difference from the Bayesian case is that we "compile out" the parameters before performing inference; also, we must also use specialized learning procedures to estimate the parameters.

Parameter tying is also used when we have a large model with repetitive structure (in addition to tying across data cases). An obvious example is the use of HMMs for arbitrary length sequences. Recently, many different kinds of "syntactic sugar" for specifying parameter tying have been devised in the UAI community. Examples include Object Oriented Bayes Nets [KP97, BW00], Probabilistic Relational Models [GFKP01], Relational Markov Models [TAK02], Dynamic Probabilistic Relational Models [SDW03], etc.[23] Below we show how we can represent these models in our proposed language. We find this much simpler than the typical presentation of PRMs etc., which often use a lot of confusing database/logic terminology and heavy notation. (See also Section 2.3.9.)

2.3.7 Generative 2D lattice MRF

Consider the model in Figure 8. This is a 2D lattice on the X variables (as in the Ising/Potts model), and then each X_i generates an observed Y_i. If the model is spatially homogeneous, the parameters are tied across all sites/edges. Let us suppose $P(Y_i|X_i)$ is conditional Gaussian with parameters μ, Σ. Since the parameters are tied, we designate them as (constant) nodes in the graph, so they can be shared. The joint is

$$P(X_{1:4}, Y_{1:4}) = \frac{1}{Z} \prod_{i \sim j} \Psi(X_i, X_j; \theta) \prod_{i=1}^{4} \mathcal{N}(Y_i|X_i; \mu, \Sigma)$$

If there are N nodes in the lattice, the model can be specified as follows.

```
N = nnodes(G);
CPD{Xtheta} = Constant(value =  2D array);
```

[22]This method of defining how to tie parameters is considerably more elegant and finer-grained than the BNT technique of defining equivalence classes between CPDs.

[23]PRMs generalize object oriented Bayes nets (OOBNs) which can only capture the "is-a" or "part-of" relations.

```
CPD{Xmu} = Constant(value =  2D array);
CPD{XSigma} = Constant(value = CovMatrix list);
for i=1:N
  for j=find(G(i+1,:))
     e = edgeNum(G, X(i), X(j));
     pot{e} = TabularPot(paramParent = Xtheta, domain=X([i j]))
for i=1:N
   e = edgeNum(G, X(i), Y(i));
   pot{e} = GaussianCPD(child = Y(i), parents = X(i),
              meanParent = Xmu, covParent = XSigma);
```

CovMatrix is a class for representing positive definite matrices. We only ever need to store upper half. Also, we allow for special cases (to save more space and reduce number of parameters): DiagonalCovMatrix and SphericalCovMatrix. If Σ is tied across conditioning cases, we just need a single CovMatrix rather than a list.

2.3.8 Discriminative relational Markov networks

A probabilistic relational model (PRM) [GFKP01] is a way of defining a joint probability distribution, using a DAG, over a set of random variables which are related in some way. For example, the variables may represent web pages, and the relations might be "has hyper-link to", or the variables might be papers, and the relation might be "cites". The need to avoid directed cycles in PRMs can prove problematic in many domains, so [TAK02] introduced the undirected analog, which they call "relational Markov networks" (RMNs). (A Markov network is the same as a Markov random field (MRF).)

We now show how to represent RMNs using our language. To be concrete, we focus on the example of classifying web pages described in [TAK02]. Consider the toy example in Figure 9(a)). We will introduce observed random variables to represent the observed words, and one hidden random variable per page to represent the topic/ category of that page. The goal is to define $P(C1, C2, C3|w1, w2, w3)$, where the C_i are the categories of each page, and the w_i are count vectors derived from a bag-of-words model, and then use this to estimate the most likely categories of the pages. The fact that we are conditioning on the w_i instead of generating them makes this a discriminative (conditional) model, which are usually work better for classification tasks than generative models [NJ02].

We will now define some rules for constructing an undirected graphical model (Markov network) from the observed relational patterns; [TAK02] calls these rules "clique templates". The first rule says that if page i contains any pointers to page j, then we add an undirected link between C_i and C_j, and parameterize this with potential ϕ_1: see Figure 9(b) for the result. This captures the intuition that pages that link to each other are often in the same category. The second rule says that if a page contains two different links, pointing to i and j, then add a link between C_i and C_j parameterized by ϕ_2: see Figure 9(c) for the result. This captures the intuition that if a page links to several other pages, the target pages are often in the same category. Finally, the third rule says that, for each page i, add an undirected link between every word in page i and the category C_i, parameterized by ϕ_3. This captures the naive Bayes/ logistic regression structure. See Figure 9(d) for the final model.

We can create this model using our simple specification language as follows.

```
% parse the data
Npages = 3;
NwordsPerPage = [1 2 3];
links = cell(1, Npages);
links{1} = [2 3];

% Make the graphical model
Nnodes = 3 + Npages + sum(NwordsPerPage);
[Xphi1, Xphi2, Xphi3, XC(1:Npages), Xw(1:sum(NwordsPerPage))] = 1:Nnodes;
CPD{Xphi1} = Constant(value = phi1);
CPD{Xphi2} = Constant(value = phi2);
CPD{Xphi3} = Constant(value = phi3);
potnum = 4;
```

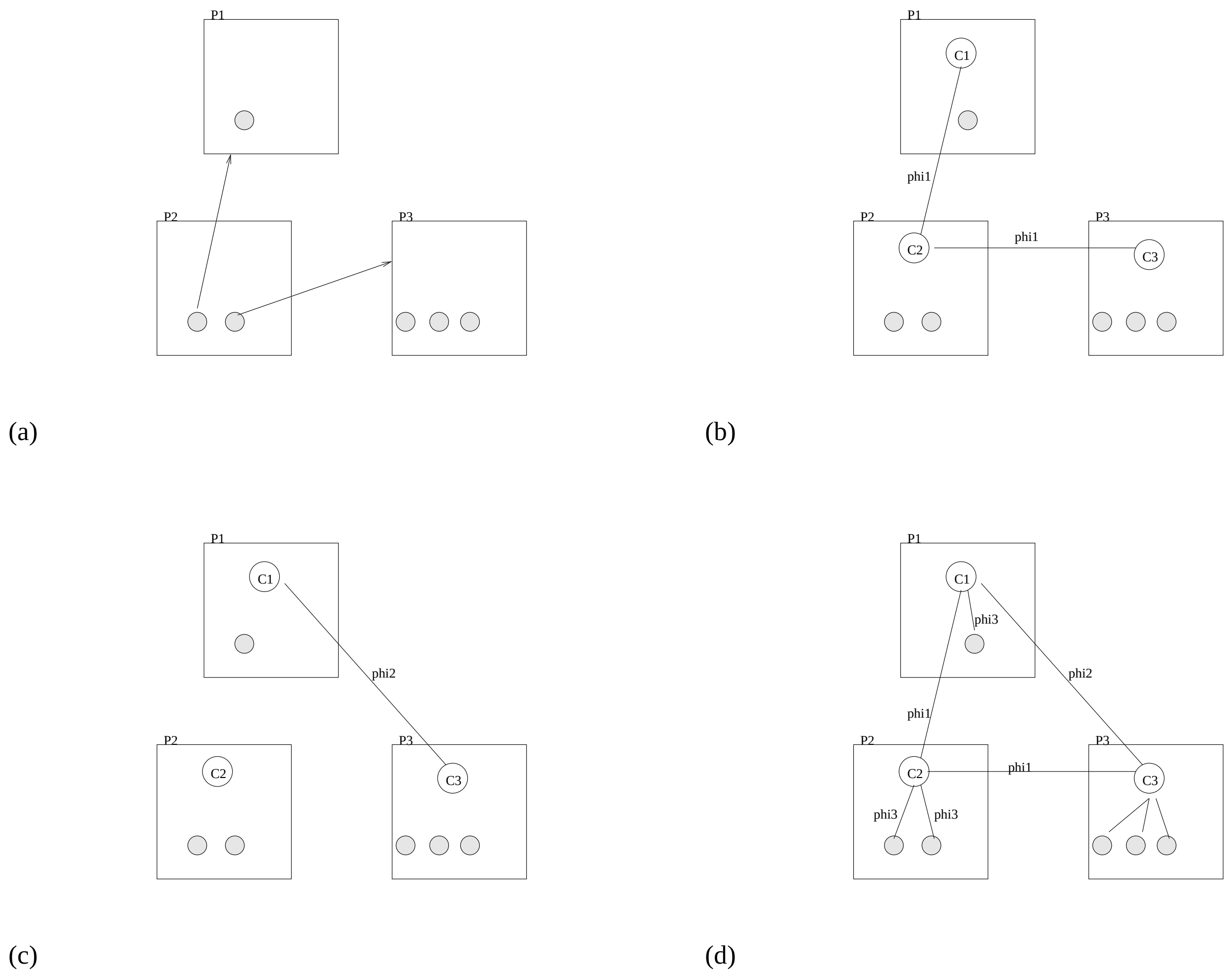


Figure 9: A relational Markov model for web page classification, based on [TAK02]. (a) The input data is 3 tiny web pages, P1 with 1 word, P2 with 2 words, and P3 with 3 words, and P2 has hyper-links to P1 and P3. (b) The first rule induces arcs 2-1 and 2-3 between latent category nodes. (c) The second rule induces arc 1-2. (d) The third rule induces naive Bayes classifier arcs between the words (shaded) and the category (unshaded).

```
for p=1:Npages
  Nlinks = length(links{p});
  % rule 1
  for l=1:Nlinks
     ptarget = links{p}(l);
     pot{potnum} = TabularPot(domain = XC([p ptarget]), paramParent=Xphi1);
     potnum = potnum + 1;
  end
  % rule 2
  for l1=1:Nlinks
     for l2=l1+1:Nlinks
        p1 = links{p}(l1);
        p2 = links{p}(l2);
        if p1 ˜= p2
          pot{potnum} = TabularPot(domain = XC([p1 p2]), paramParent=Xphi2);
          potnum = potnum + 1;
        end
     end
  end
  % rule 3
  for w=1:NwordsPerPage(p)
     pot{potnum} = TabularPot(domain = [XC(p) Xw(p,w)], paramParent=Xphi3);
     potnum = potnum + 1;
  end
end
```

2.3.9 Structural/ relational uncertainty

In PRMs and RMNs, the relational structure is usually assumed to be known (observed). Given a fixed relational "skeleton", the models merely specify a way of tying parameters.

In many problems, such as computer vision, the relations are not observed, and need to be inferred from the non-relational (flat) data, a significantly harder problem. It is true that in an image we can observe that one pixel is next to another (and hence the image is not a "bag of pixels" or even a "vector of pixels"), but this is only useful for low level image processing problems (which can model this relation using a lattice MRF, as in Section 2.3.7). More interesting relations are the ones between latent variables, such as "the person is next to the tree"; obviously these relations have to be inferred, since the image does not come with labels specifying the location (or indeed existence) of objects.

In [PMM$^+$02], they perform MCMC over the space of possible worlds (skeletons): they alternate between sampling the number of objects and relationships between them, and then performing inference in the resulting (fixed) graphical model. Zhu et al. [TZ02, BZ03] is doing interesting work combining heuristic proposal distributions with MCMC for image interpretation using probabilistic models which are much more general than graphical models. It would be interesting if we could come up with more efficient (deterministic) algorithms, which can integrate over different structures, c.f., the inside-outside algorithm [JM00], which can sum over an exponential number of parse trees, as defined by a stochastic context-free grammar. (SCFGs are strictly more general than fixed-sized graphical models.) However, this is (way) beyond the scope of the gR project.

2.3.10 Dynamic Bayesian networks (DBNs)

It is easy to specify a model which can handle variable-sized data (eg., sequences of different lengths, images of different sizes, varying numbers of objects) by manually tying parameters, as we saw in Section 2.3.8. However, this requires that the user manually creates the "unrolled" model. There are several problems with this:

- For online inference (Section 3.5), the program will be (implicitly) unrolling the model for us, and hence needs to know what the basic unit to repeat is.

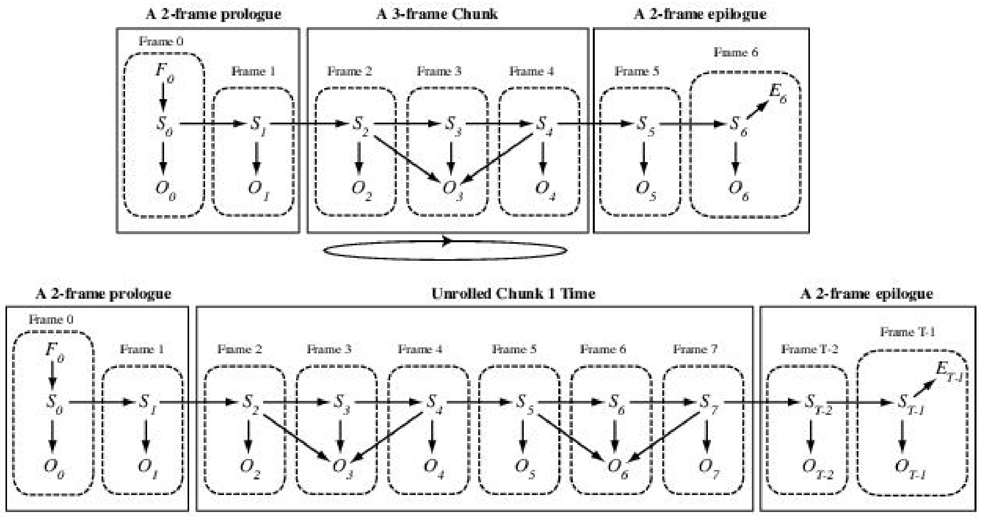


Figure 10: A dynamic Bayes net (DBN). Top: specification of the model, with prolog, repetitive chunk, and epilog. Bottom: the model unrolled for 2 chunks. From [BB03].

- For repeated batch inference, where the batches have different size (e.g., many sequences of different length), we would like to avoid having to unroll the model and retriangulate each time (if using the junction tree algorithm).

A standard time-sliced dynamic Bayes net can be defined by specifying two graphs, one encoding the intra-slice topology, and one encoding the inter-slice topology. A more general notion, which allows higher-order Markov models, backwards-in-time arcs, and includes the possibility of a prolog and epilog with different structure, can be defined [BB03]: see Figure 10 for an example. (For online inference, the epilog does not exist.) Given such a model, it is possible to determine the basic repetitive unit, and, if necessary, to triangulate it only once (see Section 3.5). Thus the DBN notation is not just syntactic sugar for parameter tying, but is actually exploited computationally. The GMTk package[24] offers an excellent C++ implementation of junction tree inference for such models, focussing on speech recognition applications.

However, we need a more general definition of DBN, since it should be possible to designate that some nodes (e.g., parameters) should not be replicated over time.[25] For example, consider the model in Figure 11. On the left is an undirected DBN template, where the parameters are drawn outside the chunk box meaning they are not to be repeated. On the right is the model unrolled for 3 slices. This is in fact a 1D conditional random field [LMP01], since all the hidden variables are conditioned on the global observations, $y_{1:3}$. That is, the model specifies

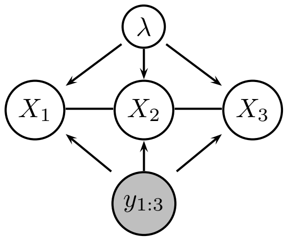
$$P(X_{1:T}|y_{1:T}) = \frac{1}{Z(y_{1:T})} \exp[\sum_t \sum_i \lambda_i' * f_i(X_{t-1}, X_t|y_{1:T})]$$

Let us suppose there are 2 feature vectors, f_1 of length 5 and f_2 of length 10. Then we can create this model using something like this:

```
pot{Xlambda(1)} = Constant(table: 5-vector);
pot{Xlambda(2)} = Constant(table: 10-vector);
pot{X12} = LogLinPot(paramParent = Xlambda(1:2), feature = f{1:2}, domain = X(1:2));
G = zeros(4,4);
G(X(1), X(2)) = 1;
```

[24]ssli.ee.washington.edu/~bilmes/gmtk/

[25]Although we can always copy a variable and give it constant dynamics, doing so may obscure the possibility of more sophisticated inference algorithms that exploit the static nature of some of the variables [TDW02]. Also, naive application of particle filters to parameters with constant dynamics can fail; see [DT03] for a possible solution.



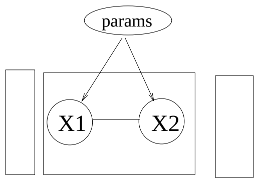


Figure 11: (Left) A 2-slice undirected DBN with stationary parameters and an empty prolog and epilog. (Right) The model unrolled for 3 slices. This is in fact a 1D conditional random field, since all the hidden variables are conditioned on the global observations, $y_{1:3}$.

```
G(X(2), X(1)) = 1;
G(Xlambda(1), X(1:2)) = 1;
G(Xlambda(2), X(1:2)) = 1;
dbn = mkDBN(pot, G, static = Xlambda(1:2));
```

3 Inference

3.1 Types of inference engine

We can roughly classify inference algorithms as follows.

3.1.1 Exact methods for models in which all hidden nodes are discrete

There has been a huge amount of work in the UAI community on exact inference for models in which all hidden nodes are discrete.[26] Exact (closed-form) inference is always possible in this case, but it is only computationally tractable for graphs with low treewidth, or with other special properties.

Here are the main types of algorithm for this class of models.

- Hugin-style message-passing on a junction tree [CDLS99, LS98].
- Variable elimination (peeling) [SDD90, AM00, Dec98].
- Recursive conditioning [AD03]. This method can trade memory for time. In practice, it can often outperform jtree-style algorithms which start thrashing on large problems.
- Brute force enumeration - naive summation over full joint; useful for debugging.

These methods fall into two main camps: those that can compute a single marginal in $O(N2^w)$ time, and those that can compute *all* (local) marginals in $O(N2^w)$ time, where w is the treewidth of the graph and N is the number of binary latent variables. Variable elimination is of the first kind, while Hugin-style two-pass message-passing algorithms are of the second. For example, variable elimination would take $O(N^2)$ time to compute all the marginals needed for learning an HMM of length N, whereas the forwards-backwards algorithm, which in this case is equivalent to the jtree

[26]This obviously excludes all Bayesian models, but includes many interesting applications, such as medical diagnosis, genetic pedigree analysis, even speech recognition (since the *observed* variables may have any kind of distribution).

algorithm [SHJ97], takes $O(N)$. (It is possible to add a second, backwards pass to variable elimination; the resulting algorithm [Coz00, KLD01], not surprisingly, ends up looking very much like the jtree algorithm.)

Why would anyone use a one-pass variable-elimination style algorithm? There are in fact several reasons:

- They are much simpler to implement.
- They make it easy to exploit local structure within the CPDs (such as "context-specific independence" [ZP99] and "causal independence" [ZY97, RD98]), which arises because the CPDs are derived from an underlying logical representation.
- They make it easy to exploit the particular pattern of evidence/query.

Recently, some of these advantages have been translated to the jtree approach using lazy junction trees [MJ99]. Nevertheless, for applications in which we make only a small number of queries before the model is modified, the variable-elimination approach may be preferable.

It is also useful to define inference engines for particular classes of models. By making assumptions about the model, it is possible to write simpler and usually faster code. For example, the quickscore algorithm for QMR [Hec89] exploits properties of the noisy-OR distribution. Also, it is possible to exactly compute the most probable (Viterbi) assignment for 2D binary Ising models based on max-flow min-cut [BK01] in $O(N^3)$ time, even though for an $N = n \times n$ 2D MRF, the optimal treewidth is $O(n)$ [LT79] (so we would expect it to take $O(2^n)$ time). Unfortunately, this method cannot compute marginals (only Viterbi assignments), and the non-binary case is NP-hard. (Also, even N^3 is too much for large lattices.)

3.1.2 Exact methods for models in which some hidden nodes are not discrete

At the current time, to the best of my knowledge, the most general class for which exact inference algorithms are known is the class of conditional Gaussian (CG) models. A Hugin-style algorithm is presented for this class in [Lau92]. This turns out to be numerically unstable; a better algorithm (but which requires more significant changes to the Hugin architecture) is described in [LJ01].

Unfortunately, exact inference in CG networks is NP-hard even for tree-structured networks [LP01]; this follows from the need to create a strong junction tree. Hence in practice even the CG class cannot be solved exactly.

If all local factors are Gaussian, the model just describes a large multi-variate Gaussian. Such models can always be solved in $O(N^3)$ time (by matrix inversion). Sparsity can often reduce this to $O(N^2)$. But for some problems even N^2 is too much, and other approximate approaches, such as thin junction trees [Pas03] or loopy belief propagation [WF99], must be used (see below).

3.1.3 Deterministic approximations in which all hidden nodes are discrete

Although discrete state models can always be solved exactly in closed form, it is often computationally intractable to do so. Therefore there has been a lot of work on approximate inference algorithms for this case. Popular algorithms include

- Mean field methods [SO01]
- Loopy belief propagation [YFW01, KFL01].
- Generalized belief propagation [YFW01] and cluster variational methods [YFW02].
- Graph cuts for Potts models [BVZ99, BK01, KZ03].
- Structured variational methods [JGJS98], which use exact inference on tractable subgraphs.
- Bounded cutset conditioning [Dar95].

Loopy BP is particularly popular, for several reasons:

- BP is very easy to implement: just run message passing on the original graphical model, and ignore the fact that it may have loops.

- BP works better than mean field for roughly the same computational cost [Wei01].
- BP is the basis of the decoding algorithm for turbo-codes and LDPC-codes, which can finally achieve the Shannon bound on communication capacity [Mac03].
- BP has some theoretical guarantees [WF99, Wei00, WF01].

3.1.4 Deterministic approximations in which some hidden nodes are not discrete

For the continuous/ Bayesian case, popular algorithms include

- Moment matching/ assumed density filtering [CDLS99, HT01] which is a sequential algorithm that recursively projects a complex posterior to a simpler form after each Bayesian update.
- Expectation propagation [Min01b]. Roughly speaking this is a batch/iterative form of assumed density filtering.
- Variational Bayes[27], as in the VIBES package.[28]. In practice, VB inference takes the form of generalized EM, where in the E step, we do inference using modified parameters (not the ML estimates).

One way to understand the difference between EP and VB is that EP locally optimizes the "right" KL-divergence, $D(P||Q)$, whereas VB globally optimizes the "reverse" KL-divergence, $D(Q||P)$ (where P is the true posterior and Q is the approximation).

3.1.5 Stochastic approximations

The main families are

- Gibbs sampling
- Metropolis Hastings
- Importance sampling - also called likelighood weighting in the UAI community.

Stochastic approximations are indifferent to whether the hidden variables are discrete or continuous, and can be applied to a large range of models. Unfortunately, importance sampling does not scale well to large spaces, and MCMC (which includes Gibbs sampling and Metropolis Hastings) can be very slow and unreliable, since in practice one often has to make a decision before convergence can be guaranteed.

One can obtain much better performance from MCMC by using a few tricks, such as

- Use deterministic approximations as proposal distributions (see e.g., [dFHSJR01, TZ02]).
- Sample many variables at once (Blocks Gibbs sampling [JKK95]).
- Make large moves through state-space (such as in the Swendsen-Wang algorithm for Potts models [BZ03]).
- Integrate out some variables using exact inference and sample the rest (Rao-Blackwellisation [CR96]).
- Combine with belief propagation, as in non-parametric BP[29].

Thus efficient sampling engines will typically have deterministic inference engines embedded inside of them.

[27]See www.variational-bayes.org for many papers, such as [Att00, GB01, Jaa01, XJR03].
[28]VIBES is currently unavailable, but is supposed to be released at www.inference.phy.cam.ac.uk/jmw39/
[29]ssg.mit.edu/nbp

3.1.6 General comments on inference engines

Sometimes we may choose to use a theoretically slower algorithm because it is simpler and hence has lower overhead (constant factors); in practice, such an algorithm may out-perform more sophisticated methods (at least on small problems). For example, the BNT hmm-inf-engine can convert many kinds of DBNs to an HMM; it then runs the forwards-backwards algorithm [Rab89], which is extremely simple and fast; finally, it derives the desired marginals from the output of FB.

Also, we may have several implementations of the same algorithm e.g., some in Matlab/R, some in C. The C code will be faster but harder to understand, modify, etc. It is easy to handle a proliferation of inference engines by using the class system: they will all have the same interface (see Section 3.2), and hence can be used interchangeably.

3.2 Things you can do with an inference engine

These are the methods that each inference engine must implement.

- `infEng = mkInfEngine(GM, [observedNodes: int list], [queryNodes: int list])`
  You may optionally specify which nodes will always be observed, and which nodes you will query. The engine may use this information to make certain optimiziations. If you do not specify them, the assumption is that any node (apart from latent variables) can be observed, and any node can be queried.

- `infEng = enterEvidence(infEng, dataCase)`
  A data case is a list of nodes, and a list of their values (which may have different types, including "missing"). All methods below assume this function has been called.

- `J:JPD = query(infEng, nodes)`
  J is the joint probability distribution (see Section 3.2.1) over the specified nodes, given the data entered previously. Note that many engines can only answer a restricted range of queries e.g., the junction tree algorithm can only compute marginals on subsets of nodes that reside within a clique (although the jtree structure can be modified to answer non-local queries [EH01]).

- `values = viterbi(infEng, [k])`
  This returns the k most probable assignments to the nodes [Nil98]. Not all engines can support this function.

- `l = logLikelihood(infEng)`
  This returns the log-likelihood of the data, with the parameters fixed to their ML/MAP estimate.

- `l = logMargLikelihood(infEng)`
  This returns the log-marginal-likelihood (the log "evidence"), integrating out parameters. This is useful e.g., for model selection. Not all inference engines will be able to compute this, or at least, not exactly. And it may be difficult to predict this in advance.

3.2.1 JPDs returned by query

JPD is a (virtual) class for representing joint probability distributions. There are several possible representations:

- TabularJPD
- GaussianJPD
- MixGaussJPD
- SampleJPD, a set of weighted samples

Each one of these sub-classes should implement the following kinds of methods (where we assume the joint is $P(X|e)$):

- `m = mode(J)`
  Compute the most probable value

- `m = mean(J, [f])`
  Compute the expected value, $m = E[X|e]$. If an optional function is specified, $m = E[f(X)|e]$.
- `q = quantiles(J)`
  Compute quantiles of the distribution.
- `J2 = marginalize(J, nodes)`
  Compute $P(X(nodes)|e)$ from $P(X|e)$.
- `plot(J)`
  Visually display (somehow) the distribution.

3.3 Applicability of inference engines

There are basically two classes of inference engines: deterministic and stochastic. Each of these engines require that the local factors support certain functionality (see Section 3.4). If the model contains a local factor that cannot implement the required method, then the corresponding inference engine cannot be applied. This may vary depending on the evidence and the query.

For example, suppose we have a model with discrete and continuous variables. If all the continuous variables are observed, then all the hidden variables are discrete, and inference is trivial (at least, if the treewidth is small). However, if some of the continuous variables are hidden, inference may not be so easy, but it depends precisely on which variables are hidden, and what the query is. For example, if the hidden continuous variables are leaves of a DAG, they can be safely removed; if the hidden continuous variables are d-separated from the query by the evidence, they can be safely removed, etc.

Hence when the user creates an inference engine and asks a query, it might not be possible to answer it, or at least, not exactly.

3.4 How inference engines are implemented

We first introduce a new class, InternalPotential, which is needed by many of the deterministic engines. Then we define what demands the two main classes of inference engines place on the local factor classes. Finally we briefly mention how some of the engines themselves are implemented (obviously there is no space to go into all of them).

3.4.1 Internal potentials

Deterministic inference engines, such as junction tree, variable elimination and loopy BP, all need to use a class that we call InternalPotential. (I would welcome suggestions for a better name!) A Potential is a way for the user to specify parameters of the model (the undirected analog of a CPD), whereas an InternalPotential is an object that is used internally by the deterministic engines for computation.

We will consider 3 kinds of InternalPotential:

- TabularInternalPotential
- GaussianInternalPotential
- MixGaussInternalPotential

An InternalPotential must support the following basic operators:

- `pot = multiplyBy(pot, pot2)`
  Element-wise multiplication of pot2 onto pot.
- `pot = divideBy(pot, pot2)`
- `pot2 = marginalizeOnto(pot, nodes)`
- `pot2 = maxMarginalizeOnto(pot, nodes)`

- `pot2 = absorbEvidence(pot, nodes, values)`

For Tables, the operations just consist of manipulating multi-dimensional arrays. Although this is easy to do, it can be quite slow[30], because of the need to perform non-sequential access to the data [Mur02b]. For Gaussians, it's best to work in canonical form (using the precision/information matrix, instead of the covariance matrix). For MixGaussInternalPot things get trickier, because mixtures of Gaussians are not closed under marginalization [Lau92, LJ01], so one needs to use "weak marginalization" (moment matching). See [CDLS99] for details.

3.4.2 Local factor methods needed by deterministic inference

To apply determinstic inference to a model with undirected edges, every user Potential must be converted to an InternalPotential. TabularPot can trivally be converted to TabularInternalPot (just by copying fields), and similarly for Gaussians and mixtures of Gaussians. LogLinPot can be converted to a TabularPot if the latent variables are discrete.

To apply determinstic infernece to a model with directed edges, we need a way to convert every kind of CPD to an InternalPotential (possibly conditioned on evidence). Every CPD for a discrete child with discrete parents can be converted to a TabularInternalPot by enumeration, although the resulting potential might be very large. However, once we have continuous variables, it may not longer be possible to exactly convert the CPD to one of these forms. We must either try to approximate the CPD by a Gaussian or mixture of Gaussians (see e.g., [Mur99]), or we must resort to stochastic methods.

Loopy belief propagation (BP) needs a way to compute messages for each potential/CPD. This is best done by converting all CPDs/Potentials to InternalPotentials, creating a factor graph [KFL01], and using the standard multiplication/marginalization operators. Sometimes there are more efficient methods for computing messages for special classes of CPD, such as noisy OR [Pea88]. Hence some CPD classes might have extra methods to help the BP engine.

3.4.3 Local factor methods needed by stochastic inference

Likelihood weighting (importance sampling) is only defined for DAGs. It just requires that, for each CPD, you are able to sample efficiently from $P(X|y)$, where X is the child and the parents y are known (instantiated).

Gibbs sampling needs a way of sampling from the Markov blanket of each node. For a DAG, this is

$$P(X_i|m_i) = P(X|pa_i) \prod_{c \in \text{children}(X_i)} P(x_c|pa_c)$$

where pa stands for parents, lower case letters represent observed quantities, and only the child X_i is hidden. Hence this product is in fact a 1-dimensional quantity. We need a way to represent and sample from this product. This requires that the CPDs can be multiplied (e.g., by converting them all to the same kind of InternalPotential) and sampled from.

3.4.4 How loopy belief propagation could be implemented

To apply LBP to a Bayes net, we can use Pearl's formulation in terms of λ and π messages [Pea88, MWJ99], but this is a bit ugly because of the asymmetry. Also, it is tricky to apply LBP to an MRF, since it is the cliques, not the nodes, that have local terms associated with them. For pairwise MRFs, it is simple to formulate the message passing rules in terms of matrix-vector multiplication [Wei00]; the pairwise Gaussian case is also easy [WF99]. Any MRF with potentials defined on larger sets of nodes can be converted into a pairwise MRF by creating mega-nodes [WF99], but this is not always desirable, since the state-space of the mega-nodes will be much larger. It is possible to derive the rules for arbitrary MRFs, but they are a bit messy. Besides, we would like to use the same code for Bayes nets and MRFs. We therefore convert both BNs and MRFs to factor graphs [KFL01, Fre03].

The basic "rules" for message passing in a factor graph are as follows. Each variable node x sends a message to each neighboring factor node f of the form

$$\mu_{x \to f}(x) = \prod_{g \neq f} \mu_{g \to x}(x)$$

[30] In BNT, about 70% of the time spent by deterministic inference engines is spent manipulating Tabular potentials.

and each factor node f sends a message to each neighboring variable node x of the form

$$\mu_{f \to x}(x) = \sum_{u \setminus x} f(u) \prod_{y \neq x} \mu_{y \to f}(y)$$

where we have assumed u is the domain of f. The messages and factors are both represented as InternalPotentials, so that we can multiply and marginalize them. (The product of all-but-one messsages can be implemented by dividing one out from the product of all.)

If the fgraph was derived from a tree, this procedure is exact. In this case, the most natural message passing order is from leaves to root (pre-order) and back (post-order). For loopy graphs, we can perform message passing in parallel.

3.4.5 How junction tree inference could be implemented

Figure 12 outlines some of the graph-theoretic routines that are needed to create a junction tree from a graph. It assumes that we create the elimination order as soon as we have the graph. However, we can imagine some alternatives, as we discuss below.

Inference engine methods are usually called in the following sequence:

1. Create the engine (call the constructor), which specifies the graph and the local factors
2. Enter evidence
3. Ask a query

The amount of work that is done at each stage is engine-specific. For example, for the jtree inference engine, we do the following:

1. Constructor: make the junction tree
2. Enter evidence: message passing
3. Ask query: marginalize clique potential

It is clear that if we are lazy, and delay making the junction tree until we know what the evidence/ query is, we can create a better tree (smaller cliques). However, building a new jtree for each query can be expensive. This is why we have the optional arguments to the constructor, that specify up-front what nodes will be observed/queried.

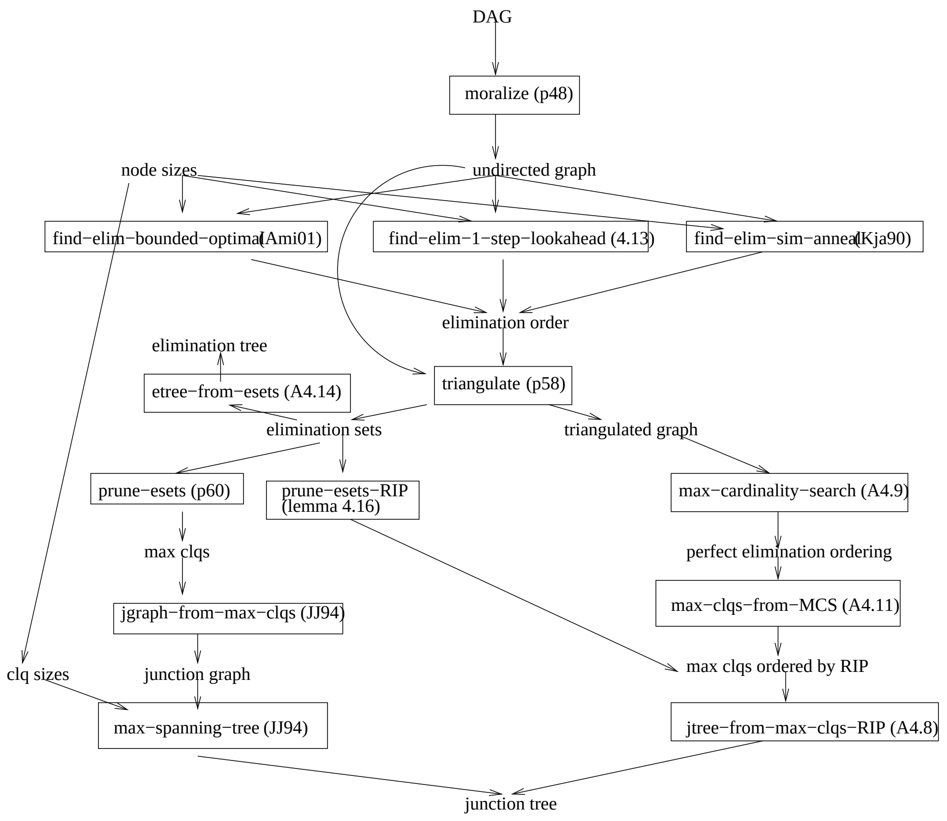
Once we have a junction tree, we can easily convert it to a factor graph (cliques become factors and separators become variables), and then apply belief propagation; the result is equivalent to the Hugin algorithm[31], but has the advantage of building on top of the BP engine, which minimizes code duplication.

3.5 Online inference

So far we have only considered batch inference. In many applications, we want to sequentially/recursively update our belief state as new evidence arrives, i.e., to compute $P(X_t|y_{1:t})$ given y_t and $P(X_{t-1}|y_{1:t-1})$ (or some sufficient statistic), where X_t represents all the hidden variables/parameters in slice/chunk t (see Section 2.3.10). This task is called filtering. Other tasks include prediction (computing $P(X_{t+k}|y_{1:t})$) and smoothing (computing $P(X_t|y_{1:T})$, which can be used for offline inference). See [Mur03] for a tutorial.

We first propose some high-level engines for performing these tasks. This makes use of an underlying FB inference engine, which implements two methods, called forwards and backwards, which we discuss below.

[31]To make the equivalence exact, we need to exploit the fact that in a jtree, separators (variables) are always connected to exactly two cliques (factors). It turns out that this simplifies the factor graph message passing algorithm, with the result that only cliques (factors) send messages, and separators (variables) just store products of messages. See [Mur02a, p157].



DAG
moralize (p48)
undirected graph
node sizes
find−elim−bounded−optimal (Ami01)
find−elim−1−step−lookahead (4.13)
find−elim−sim−anneal (Kja90)
elimination order
elimination tree
etree−from−esets (A4.14)
triangulate (p58)
elimination sets
triangulated graph
prune−esets (p60)
prune−esets−RIP (lemma 4.16)
max−cardinality−search (A4.9)
max clqs
perfect elimination ordering
jgraph−from−max−clqs (JJ94)
max−clqs−from−MCS (A4.11)
clq sizes
junction graph
max clqs ordered by RIP
max−spanning−tree (JJ94)
jtree−from−max−clqs−RIP (A4.8)
junction tree

Figure 12: Creating a junction tree from a graph, from Appendix B of [Mur02a]. There are essentially two approaches, shown in the left and right columns, corresponding to elimination sets/ max-spanning-trees and max-cardinality-search. (BNT uses the approach on the left.) Boxes represent functions; quantities outside of boxes are variables. Every function has a citation: pX refers to page X of [CDLS99]; AX refers to algorithm X of [CDLS99]; [Ami01, Kja90, JJ94a] are papers from the bibliography. We have ignored the issue of constrained (strong) triangulation for simplicity (see p133 of [CDLS99]).

3.5.1 High level online inference engines

Given an underlying FB infererence engine, which implements the basic forwards and backwards operators, we can build filters, smoothers (fixed-interval and fixed-lag), and predictors in a uniform way [Mur02a, LBN96]. We can also easily implement log-space fixed-interval smoothing [BMR97, ZP00] using these primitives, which is essential for applying EM to long sequences. (This algorithm is implemented in GMTk but not BNT.)

The interface to the filter engine is as follows.

```
infEng = filterEngine(FBEngine);
for t=1:infty
  infEng = enterEvidence(infEng, nodes, values); % update with yt
  J: JPD = query(infEng, nodes); % any subset of X(t-1:t|y(1:t))
end
```

The interface to the fixed-lag smoother engine is as follows.

```
infEng = fixedLagSmootherEngine(FBEngine, lag = k);
for t=1:k
  infEng = enterEvidence(infEng, nodes, values); % update with yt
end
for t=k+1:infty
  J: JPD = query(infEng, nodes); % any subset of X(t-k:t|y(1:t))
end
```

The interface to the fixed-interval smoother engine is as follows.

```
infEng = offlineSmootherEngine(FBEngine);
infEng = spaceEfficientOfflineSmootherEngine(GM, Nislands = log(T));
infEng = enterEvidence(infEng, nodes, values); % y(1:T)
J: JPD = query(infEng, nodes); % any subset of X(t-1:t|y(1:T)) for any t
values = viterbi(infEng, k);
```

3.5.2 The forwards and backwards operators

- $f_{t|1:t}$ `= fwd(infEng,` $f_{t-1|1:t-1}$`, evidence, [nodes])`
  This updates the filtered belief state f given the new evidence on the specified nodes. Typically we assume that the same nodes are observed at every time step, but this need not be the case. If the evidence is omitted, this function can be used for forecasting, i.e., computing $P(X_{t+k}|y_{1:t})$. For the initial time-step ($t = 1$), $f_{t-1|1:t-1}$ is just a uniform prior.
- $b_{t|1:T}$ `= back(infEng,` $b_{t+1|1:T}, f_{t|1:t}$)
  This combines the backwards smoothed belief state with the forwards filtered belief state. For the final time-step ($t = T$), $b_{t+1|1:T}$ is just a uniform prior. Note that some engines may not be able to implement this, e.g., it is inefficient to do so for particle smoothers [IB98b]. Also, note that b is a proper probability distribution, not a conditional likelihood. That is, this is like the the $\alpha - \gamma$ algorithm rather than the $\alpha - \beta$ algorithm for HMMs, or like the RTS-Kalman smoother, not a two-filter smoother, for LDSs (see [Mur02a] for a discussion).

In order to support the Viterbi algorithm for discrete state-spaces, we will need two versions of the forwards/backwards operators, which use sum-product and max-product. This can be specified as an optional argument when the engine is created.

3.5.3 Representation of the belief state

A belief state is a sufficient statistic for inference, i.e., a joint probability distribution over the interface of the DBN [BB03]. (For simple DBNs, the (forward) interface is all the nodes in a slice with outgoing arcs into the next slice.) We can use any of the representations in Section 3.2.1 to represent the belief state. However, since the interface can be quite large, we may want to represent the belief state using a junction tree, as in [Pas03], or approximately as a product of marginals, as in the BK algorithm [BK98b].

3.5.4 Possible implementation of the forwards and backwards operators

For discrete state spaces, common algorithms include

- HMM [Rab89].
- 1.5Jtree [Mur02a, BB03].
- BK [BK98b, BK98a].

For continuous state spaces, common algorithms include

- Kalman [Min99]
- ExtendedKalman [WN01]
- UnscentedKalman [WdM01]
- ParticleFilter [AMGC02, DdFG01]
- UnscentedParticleFilter [vdMDdFW00]

Many of these algorithms (forwards operator only) are implemented in the excellent Matlab package REBEL[32].

For hybrid discrete-continuous state spaces, common algorithms include

- ParticleFilter [IB98a]
- GaussianSumFilter [Min02]

4 Learning

There are many different kinds of learning. We can distinguish between the following "axes":

- Parameter learning or structure learning (model selection).
- Frequentist or Bayesian. A frequentist tries to learn a single best parameter/ model (point estimate). In the case of parameters, this can either be the maximum likelihood (ML) or the maximum a posteriori (MAP) estimate. In the case of structure, it must be a MAP estimate (or some kind of penalized likelihood), since the ML estimate would be the fully connected graph. By contrast, a Bayesian tries to learn a distribution over parameters/ models. Note: Bayesian parameter learning is equivalent to inference in a fixed GM from a set of data cases, so we will not discuss it further.
- Fully observed or partially observed. Partially observed refers to the case where there is missing data and/or latent variables. Learning in the partially observed case is much harder; the likelihood surface is multimodal, so one usually has to settle for a locally optimal solution, obtained using EM or gradient methods. If we adopt a Bayesian approach, things are arguably worse, since latent variables often make the model unidentifiable; since many of the modes might be equivalent, representing the posterior in a compact way is tricky (the label switching problem [Ste00]).
- Directed or undirected model. It is easy to do parameter learning in the fully observed case for directed models (BNs), because the problem decomposes into a set of local problems, one per CPD; in particular, inference is not required. However, parameter learning for undirected models (MRFs), even in the fully observed case, is hard, because the normalizing term Z couples all the parameters together; in particular, inference is required. Conversely, structure learning in the directed case is harder than in the undirected case, because one needs to worry about avoiding directed cycles.

[32]choosh.ece.ogi.edu/rebel/index.html

- Causal or non-causal model. If one wants to make causal interpretations of the model, one must realize that is only possible to identify a DAG up to Markov equivalence unless one has interventional data. Hence the learning algorithm should optionally be able to accept annotation specifying if the data is observational or interventional [CY99].[33] See [Pea00, SGS00] for discussion.
- Static or dynamic model. Most techniques designed for learning static graphical models also apply to learning dynamic graphical models (DBNs and dynamic chain graphs), since a DBN is just a BN with tied parameters/structure.
- Offline or online. Offline learning refers to estimating the parameters/ structure given a fixed batch of data. Online learning refers to sequentially updating an estimate of the parameters/ structure as each data point arrives. (Bayesian methods are naturally suited to online learning.) Note that one can learn a static model online and a dynamic model offline; these are orthogonal issues. If the training set is huge, online learning might be more efficient than offline learning. Often one uses a compromise, and processes "mini batches", i.e., sets of training cases at a time.
- Discriminative or not. Discriminative training is very useful when the model is going to be used for classification purposes [NJ02]. In this case, it is not so important that each model be able to explain/ generate all of the data; it only matters that the "true" model gets higher likelihood than the rival models. Hence it is more important to focus on the differences in the data from each class than to focus on all of the characteristics of the data. Typically discriminative training requries that the models for each class all be trained simultaneously (because of the sum-to-one constraint), which is often intractable. Various approximate techniques have been developed. Note that discriminative training can be applied to parameter and/or structure learning.
- Active or passive. In supervised learning, active learning means choosing which inputs you would like to see output labels for, either by selecting from a pool of examples, or by asking arbitrary questions from an "oracle" (teacher). In unsupervised learning, active learning means choosing where in the sample space the training data is drawn from; usually the learner has some control over where it is in state-space, e.g., in reinforcement learning. The control case is made harder because there is usually some cost involved in moving to unexplored parts of the state space; this gives rise to the exploration-exploitation tradeoff. In the context of causal models, active learning means choosing which "perfect interventions" to perform. See e.g., [TK01, SJ02].

Despite the variety and complexity of the problem, it is possible to come up with a single, uniform interface for all of these approaches, namely:

```
out = learn(learnEngine, dataCases);
```

All the algorithm-specific parameters are buried inside the learning engine, which also contains the initial GM (if needed). The form of the output is engine-specific, as follows:

- In the case of parameter learning, the output is a graphical model of the same form as the input, but with modified parameters.
- In the case of frequentist structure learning, the output is a new GM (new structure and parameters). If the algorithm is capabable of it, the resulting GM may have new, extra hidden variables [ELFK00].
- In the case of Bayesian structure learning, the output is some kind of distribution over GMs. This will be an abstract data type, similar to a JPD over graphs. From this, we can compute the marginal probabilities of features (e.g., "what is the probability that there is an edge $i \to j$ in the graph"), or the mode (most probable model), etc.
- In the case of constraint-based structure learning (such as the PC algorithm), the output is a PDAG (partially directed acyclic graph), that identifies the equivalence class, and annotates possible confounds.

We discuss some possible engines below.

[33] A perfect intervention corresponds to setting a node to a specific value, and then cutting all incoming links to that node, to stop information flowing upwards. For example, consider the 2 node BN where smoking $\to$ yellow-fingers; if we observe yellow fingers, we may assume it is due to nicotine, and infer that the person is a smoker; but if we paint someone's fingers yellow, we are not licensed to make that inference, and hence must sever the incoming links to the yellow node, to reflect the fact that we forced yellow to true, rather than observed that it was true. A real-world example of a perfect intervention would be choosing which gene to knock out.

4.1 Parameter learning

4.1.1 Directed models

For fully observed DAG models (Bayes nets), the (penalized) log-likelihood decomposes into a sum of local terms, one per CPD (ignoring the parameter tying issue for now). Hence we can just pass the data into each CPD, and ask it to compute its own ML/MAP estimates (see Section 4.1.2). This method finds a global optimum, so the initial parameter estimates can be arbitrary.

For learning DAGs in the partially observed case, we can either use conjugate gradient descent [BKRK97], or EM [Lau95], or both [SRG03]. These methods find a local optimum, and hence require good initial parameter estimates. These methods require an inference engine, and may take optional parameters, such as the maximum number of iterations, and a convergence test/ threshold. Also, for incremental EM (sometimes called "online EM") [NH98, TMH99], we can specify a batch size, i.e., how many data cases to process before taking a step in parameter space. We may also want to employ some techniques to avoid local minima, such as data perturbation [ENFS02] or deterministic annealing [RR01, UN98].

Discriminative learning of generative models is tricky. It may be better to train a conditional density model (see e.g., [LMP01]).

To summarize, we have the following constructors:

- `BN-fullobs-learn-engine(GM)`
- `BN-EM-learn-engine(GM, infEng, maxNumIter, convThresh, batchSize)`
- `BN-CG-learn-engine(GM, infEng, maxNumIter, convThresh, batchSize)`

4.1.2 CPD methods needed by parameter learning

Each CPD stores its expected sufficient statistics (ESS) needed for parameter estimation.

1. `cpd = initESS(cpd)`
   Initialize the ESS (e.g., set counts to 0).
2. `cpd = updateESS(cpd, J: JPD )`
   JPD is over nodes in the CPD's domain. From this, the ESS can be extracted. If the domain is fully observed, the JPD will just be a delta function.
3. `cpd = estParams(cpd)`
   If parameter are specified as constants, this computes the maximum likelihood estimate. If parameters are specified as parent nodes (containing the prior), this computes MAP estimate.

The CPD has finite ESS iff it is in the exponential family. In this case, there is sometimes (e.g., for tabular and Gaussian) a closed form solution to the MLEs (although things can get tricky even in the Gaussian case if parameters are tied across states [Mur98]). If there is no closed form solution, we can use gradient methods, e.g., IRLS for fitting a softmax [JJ94b]. (If the global maximum is not obtained in the M-step, this is called generalized EM (GEM) rather than EM.)

If all the parameters of a CPD are tied in the same way, then we simply pool the ESS from each of the nodes that share the CPD. For example, when estimating the transition matrix of an HMM [Rab89], the ESS are pooled over time slices:

$$\xi(i,j) = \sum_{t=2}^{T} P(X_{t-1} = i, X_t = j | y_{1:T}).$$

In general, EM proceeds as follows:

```
for each CPD c
  CPD{c} = initESS(CPD{c})
end
```

```
% E step
For each data case e
  enter evidence e
  for each adjustable node i
    J = query(P(X_i, Pa(X_i) | e))
    c = CPD for node i
    CPD{c} = updateESS(CPD{c}, J)
  end
end

% M step
for each CPD c
  CPD{c} = estParams(CPD{c});
end
```

If some of the parameters of a CPD are tied in different ways (e.g., the covariance of a Gaussian is tied across time, but the mean is not), things get harder.[34]

4.1.3 Undirected models

For fully observed models, we have the following cases:

- For decomposable models with fully-parameterized (tabular) clique potentials and maximal cliques, we can write down the MLEs in closed form (c.f., the case for DAGs) [Jor03].
- For arbitrary (not necessarily decomposable) undirected models with fully-parameterized (tabular) clique potentials and arbitrary cliques, we can use iterative proportional fitting (IPF) to find the MLEs [JP95]. This uses an inference engine for efficiency.
- For arbitrary undirected models with arbitrary (log-linear) clique potentials and arbitrary cliques, we can use generalized iterative scaling (GIS [DR72]), improved iterative scaling (IIS [Ber97]), or conjugate gradient methods. Recently CG methods have been found to be signicantly faster than IIS or GIS [Min01a, SP03]. All of these methods need an inference engine (assuming the graph structure is non trivial).

In the partially observed case, we can combine the above methods with EM; see [Lau95, WSZ02] for details.

4.2 Structure Learning

All the structure learners can take as input a set of constraints, which specify arcs that must/must not be present. Bayesian learners can also taken in priors over models.

4.2.1 Constraint-based methods

For constraint-based methods, such as the PC algorithm [SGS00, Pea00], the engine takes as input a conditional independency tester (itself an object). The conditional independency tester may use the "kernel trick" to handle data that is not discrete and not Gaussian [BJ03]. The output is a partially directed acyclic graph, that identifies the equivalence class, and annotates possible confounds. For example, we may write something like this:

```
condIndepTest = Ztest(alpha = 0.05);
constraints.forbiddenEdges = { [1 2], [3 10] };
constraints.mandatoryEdges = { [1 5]};
structLearnEng = learn_struct_PC(data, condIndepTest, constraints);
pdag = learn(structLearnEng);
```

[34] See e.g., ssli.ee.washington.edu/∼bilmes/gmtk for a description of how GMTk handles the sub-Gaussian parameter tying case.

4.2.2 Search-based methods

Search-based methods can search over various spaces, such as

- DAGs
- Equivalence classes of DAGs [Chi02]
- Total orderings of DAGs [FK03]
- Undirected decomposable models [GGT00]

They can also use various search algorithms, such as

- Greedy hill-climbing, possibly with multiple restarts
- MCMC or simulated annealing.[35]
- Genetic algorithms

They also need a scoring function, such as

- Marginal likelihood
- BIC/ MDL
- Cross-validated likelihood

The scoring function may use the "kernel trick" to handle data that is not discrete and not Gaussian [BJ03].
In the partially observed case, computing the score typically requires an inference algorithm. Options include

- Gibbs sampling [Chi95]
- Variational Bayes [BG02]
- Expected BIC score [Fri97]
- Laplace approximation [CH97]

Putting this all together, the constructor looks like this:

```
learnEng = structSearchEngine(GM, constraints, nbrFn, searchAlgo, scoreFn);
```

where

- The nbrFn is an object which can generate neighbors in the appropriate search space. This is an object because it might have internal state, e.g., we can easily generate all graphs that differ from the current one by adding, deleting or reversing a single arc, but we may then want to check for acyclicy; [GC01] shows how to do this in $O(1)$ time using an auxiliary data structure called an ancestor matrix, which should be stored inside nbrFn for modularity reasons.
- The scoreFn is an object that can compute the score; this may have an inference engine embedded inside of it, and a model prior.

For example, to implement hill climbing over DAG space using a variational Bayes approximation to the marginal likelihood (to handle partial observability), we may write

[35]To get some idea of the computational (in)feasability of applying MCMC to large structure learning problems, consider the experiments in [GC01]. They ran MCMC over the 3,781,503 DAGs on 6 binary nodes (many of which are Markov equivalent), using a fully observed data set of 1,841 cases. They used 100,000 iterations with no burn-in, but it seemed to converge after "only" 10,000 iterations. To scale up to larger problems, one will need smarter tricks, such as Rao-Blackwellisation [FK03].

```
nbrFn = nbrsDAG(GM.G); % makes the ancestor matrix in O(N) time
infEng = variationalBayesInfEng(GM);
modelPrior = uniformPriorOnDags(nnodes = length(GM.G));
scoreFn = marginalLikelihoodScoreFn(infEng, modelPrior);
searchAlgo = hillClimbing(nrestarts = 10);
learnEng = structSearchEngine(GM, constraints, nbrFn, searchAlgo, scoreFn);
output = learn(learnEng, dataCases);
```

The result is either the best model found, or all the models found on each restart.

As another example, to implement Rao-Blackwellised MCMC over node orderings [FK03], using the marginal likelihood score given by the K2 algorithm (assuming complete data), we may write

```
nbrFn = nbrsNodeOrder(length(GM.G));
modelPrior = uniformPriorOnDags(nnodes = length(GM.G));
scoreFn = marginalLikelihoodK2(modelPrior, maxFanIn = 3);
searchAlgo = MCMCsearch(burnin = 1000, maxNsteps = 10000);
learnEng = structSearchEngine(GM, constraints, nbrFn, searchAlgo, scoreFn);
output = learn(learnEng, dataCases);
```

The result is a distribution over node orderings, which can be converted into a distribution over graphs or graph features.

Note that if one is only interested in using the resutling model as a black-box density estimator (as opposed to trying to interpret/visualize it in some way), it would be much more efficient to use e.g., mixtures of trees [MJ00].

4.3 Scaling up to large data sets

For large data sets, we will need ways to iterate over large databases, without loading all the data in at once. (Incremental EM [NH98, TMH99] is also useful in this case.) We will also need smart algorithms and data structures, such as those developed/described by Andrew Moore's Auton project[36]. In this case, R would simplify provide a wrapper for code written in a more efficient language. However, it is always good to have an R prototype first!

5 Decision making

Decision networks (influence diagrams) can be solved by message passing on a junction tree or elimination tree [CDLS99]. All we need is to create a new kind of InternalPotential, called a UtilityInternalPotential, which has two parts, representing the probabilistic and utility components. Then we can create a slightly modified version of the jtree inference engine.

Unfortunately, exact solutions to influence diagrams are usually intractable to compute because of the no-forgetting assumption. This assumption means that decision nodes later in the sequence must be eliminated after all the previous actions/observations. The resulting constrained elimination ordering gives rise to large cliques.

LIMIDs (Limited information influence diagrams [LN01]) allow one to only include a subset of the information arcs, thus lifting the no-forgetting assumption. It is then possible to make an algorithm that can efficiently find a locally optimal solution to a LIMID. (If no information arcs are removed, the solution is the global optimum.) The LimidEngine uses the junction tree algorithm as a subroutine.

Influence diagrams are closely related to factored POMDPs (partially observed Markov decision processes). In particular, an ID is essentially a finite-horizon factored POMDP, in which the world state does not change unless an action is taken (the transition matrix is the identity matrix). Optimal solutions to POMDPs require solving a belief-state MDP (Markov decision process). The very fact that we are using a belief state means we are implicitly using the no-forgetting assumption. Not surprisingly, then, optimal solutions to POMDPs are NP-hard, both because solving the belief-state MDP is hard, and because computing the belief state (filtering) is hard. A natural solution to the first problem is to use finite lookahead, as in a receeding-horizon control (see e.g., [RN02, p.630]). A natural solution to the second problem is to use approximate belief-state filtering (see Section 3.5).

[36]www.autonlab.org

Another approach to approximate planning is to cast it as an inference problem [Zha98, XY01, Att03]: the goal is to infer the sequence of decisions that realizes the goal in T steps (the goal node is observed to be true at time T); we can then do binary search over T. The advantage of this is that we have well-established approximate inference techniques. However, it is limited to goal-achievement, which is just a special case of utility maximization.

There is a very large AI literature on decision theoretic planning [BDH99, Bly99], although it usually assumes complete observability. There has been some interest in using 2-slice DBNs to represent factored MDPs [BDG01], but currently this is little more than a representational device: they do not use GM inference or learning algorithms.

Obviously we are not proposing to implement all of these algorithms in R. However, it is useful to define a generic interface to any decision engine:

- `decEng = decisionEngine(GM)`
  The constructor might do an arbitrary amount of work. For example, it might optimally solve the POMDP, or do nothing, and wait to see what the evidence is.
- `a = chooseAction(decEng)`
  Pick the best action based on the current information state.
- `decEng = enterEvidence(decEng, nodes, values)`
  We specify the action we just took, and any resulting observations (outcomes).

6 Feasability of proposed design

The proposed design might be considered too ambitious. However, I think all of this could be implemented in about 4 person-years. The reason I say this is that I have already implemented about half of this proposal in BNT, and it only took me, working (mostly) alone and half-time, about 4 years. The result is about 35,000 lines of code. The reason for the fairly rapid development is because I used a very high-level language (Matlab), which liberates the mind from pesky details like memory management, and enables one to write clear, concise code. R is in fact an even more powerful language than Matlab, and has more relevant existing packages to build on top of, which should further decrease the development time and size of the code.

As another data point, Intel's Russian research lab has been developing a C++ version of BNT called PNL[37] since January 2002. With a team of about 7 full-time programmers, they have so far implemented about 25% of the functionality proposed here (about 53,000 lines of code). This suggests that it would take about $7 \times 4 \times 18/12 = 42$ person-years to implement this whole proposal in C++, i.e., about 10 times slower than using a high-level language like R. Also, the resulting code would be much larger and harder to maintain. (Indeed, the 80/20 rule[38] argues that the vast majority of the code should be written in a high-level language, with only the bottlenecks being rewritten in a faster language, like C. Although it is standard practice in industry to write everything in C++ or Java[39], this is clearly an inefficient use of resources.)

Another issue is how well the design can be implemented in parallel by a "friendly anarchy" of developers. In fact, I think the design is very amenable to distributed implementation, since the class hierarchy is broad and shallow, with no inheritance (and hence no inter-dependence). For example, it is easy to add a new engine (for inference, learning or decision making) independently of everyone else. The same goes for local factors, graphical models, JPDs, internal potentials, etc. In fact the only "closed classes" are not in fact classes, but data structures, such as Graph, CovMatrix, DataCase, etc. Of course, development time does not necessarily decrease as the number of developers increases [Bro95], but this project, by software engineering standards, is quite small and very-well defined, so the prospects are good.

[37] www.intel.com/research/mrl/pnl

[38] The 80/20 rule says that 80% of the time is spent in 20% of the code.

[39] There are certain exceptions. For example, ITA software (which powers orbitz.com) develops most of their code in Lisp. Also, the engine behind Yahoo!Store is mostly written in Lisp (see www.paulgraham.com/paulgraham/avg.html). One advantage of Lisp over other high-level languages is that there are several good compilers for it (ITA uses the CMU CLisp compiler). (OCaml also has an excellent compiler, but does not seem to be as popular for industrial applications, despite the benefits of type-safety.) The lack of native-code compilers is clearly a disadvantage of R and Matlab. See www.ai.mit.edu/∼murphyk/Software/which_language.html for further discussion of various high-level programming languages.

7 Bibliography

Note: citations are to (what I consider) the one or two most useful papers on a topic (typically the most recent), and are not necessarily to the original papers.

References

[AD03] David Allen and Adnan Darwiche. Optimal time-space tradeoff in probabilistic inference. In *Intl. Joint Conf. on AI*, 2003.

[AM00] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Trans. Info. Theory*, 46(2):325–343, March 2000.

[AMGC02] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking. *IEEE Trans. on Signal Processing*, 50(2):174–189, February 2002.

[Ami01] E. Amir. Efficient approximation for triangulation of minimum treewidth. In *Proc. of the Conf. on Uncertainty in AI*, 2001.

[Att00] H. Attias. A variational Bayesian framework for graphical models. In *NIPS-12*, 2000.

[Att03] Hagai Attias. Planning by probabilistic inference. In *AI-Stats*, 2003.

[BB03] Jeff Bilmes and Chris Bartels. On triangulating dynamic graphical models. In *Proc. of the Conf. on Uncertainty in AI*, 2003.

[BDG01] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 2001.

[BDH99] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *J. of AI Research*, 1999.

[Ber97] A. Berger. The improved iterative scaling algorithm: A gentle introduction. Technical report, CMU, 1997.

[BFGK96] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-Specific Independence in Bayesian Networks. In *Proc. of the Conf. on Uncertainty in AI*, 1996.

[BG02] M. Beal and Z. Ghahramani. The Variational Bayesian EM Algorithm for Incomplete Data: with Application to Scoring Graphical Model Structures. In *Bayesian Statistics 7*, 2002.

[BJ03] F. R. Bach and M. I. Jordan. Learning graphical models with Mercer kernels. In *Advances in Neural Information Processing Systems 15*, Cambridge, MA, 2003. MIT Press.

[BK98a] X. Boyen and D. Koller. Approximate learning of dynamic models. In *NIPS-11*, 1998.

[BK98b] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Proc. of the Conf. on Uncertainty in AI*, 1998.

[BK01] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in computer vision. In *Third International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, 2001. C++ software at above url.

[BKRK97] J. Binder, D. Koller, S. J. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29:213–244, 1997.

[Bly99] Jim Blythe. An overview of planning under uncertainty. *Lecture Notes in Computer Science*, 1600:85–??, 1999.

[BMR97] J. Binder, K. Murphy, and S. Russell. Space-efficient inference in dynamic probabilistic networks. In *Intl. Joint Conf. on AI*, 1997.

[Bro95] Frederick Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995. 2nd edition.

[BVZ99] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. In *ICCV (1)*, pages 377–384, 1999.

[BW00] O. Bangso and P. Wuillemin. Top-down construction and repetitive structures representation in Bayesian networks. In *FLAIRS*, 2000.

[BZ03] A. Barbu and S-C Zhu. Graph partition by swendsen-wang cuts. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 2003.

[CDLS99] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[CH97] D. Chickering and D. Heckerman. Efficient approximations for the marginal likelihood of incomplete data given a Bayesian network. *Machine Learning*, 29:181–212, 1997.

[Chi95] S. Chib. Marginal likelihood from the Gibbs output. *J. of the Am. Stat. Assoc.*, 90:1313–1321, 1995.

[Chi02] David Maxwell Chickering. Learning equivalence classes of Bayesian-network structures. *Journal of Machine Learning Research*, 2:445–498, February 2002.

[Coz00] F. Cozman. Generalizing variable-elimination in Bayesian Networks. In *Workshop on Prob. Reasoning in Bayesian Networks at SBIA/Iberamia*, pages 21–26, 2000.

[CR96] G Casella and C P Robert. Rao-Blackwellisation of sampling schemes. *Biometrika*, 83(1):81–94, 1996.

[CY99] G. Cooper and C. Yoo. Causal discovery from a mixture of experimental and observational data. In *Proc. of the Conf. on Uncertainty in AI*, 1999.

[Dar95] A. Darwiche. Conditioning algorithms for exact and approximate inference in causal networks. In *Proc. of the Conf. on Uncertainty in AI*, 1995.

[Daw81] A. P. Dawid. Some matrix-variate distribution theory: some notational considerations and a bayesian application. *Biometrika*, 68:265–274, 1981.

[DdFG01] A. Doucet, N. de Freitas, and N. J. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer Verlag, 2001.

[Dec98] R. Dechter. Bucket elimination: a unifying framework for probabilistic inference. In M. Jordan, editor, *Learning in Graphical Models*. MIT Press, 1998.

[dFHSJR01] N. de Freitas, P. Hjen-Srensen, M. I. Jordan, and S. Russell. Variational MCMC. In *Proc. of the Conf. on Uncertainty in AI*, 2001.

[DR72] J. Darroch and D. Ratcliff. Generalized iterative scaling for log-linear models. *Annals of Math. Statistics*, 43(5):1470–1480, 1972.

[DT03] A. Doucet and V.B. Tadic. Parameter estimation in general state-space models using particle methods. *Annals of the Institute of Statistical Mathematics*, 2003.

[EH01] T. El-Hay. Efficient methods for exact and approximate inference in graphical models. Master's thesis, Hebrew Univ., Dept. Comp. Sci., 2001.

[ELFK00] G. Elidan, N. Lotner, N. Friedman, and D. Koller. Discovering hidden variables: A structure-based approach. In *Advances in Neural Info. Proc. Systems*, 2000.

[ENFS02] G. Elidan, M. Ninion, N. Friedman, and D. Schuurmans. Data perturbation for escaping local maxima in learning. In *AAAI*, 2002.

[FK03] N. Friedman and D. Koller. Being Bayesian about network structure. *Machine Learning*, 50:95–126, 2003.

[Fre03] B. Frey. Extending factor graphs so as to unify directed and undirected graphical models. In *Proc. of the Conf. on Uncertainty in AI*, 2003.

[Fri97] N. Friedman. Learning Bayesian networks in the presence of missing values and hidden variables. In *Proc. of the Conf. on Uncertainty in AI*, 1997.

[FS03] B. Fischer and J. Schumann. Generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, 2003.

[GB01] Z. Ghahramani and M. J. Beal. Graphical models and variational methods. In M. Opper and D. Saad, editors, *Advanced mean field methods*. MIT Press, 2001.

[GC01] P. Giudici and R. Castelo. Improving Markov chain Monte Carlo model search for data mining. *Machine Learning*, 2001. To appear.

[GFKP01] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*. Springer-Verlag, 2001.

[GFSB02] A. Gray, B. Fischer, J. Schumann, and W. Buntine. Automatic Derivation of Statistical Algorithms: The EM Family and Beyond. In *Advances in Neural Info. Proc. Systems*, 2002.

[GGT00] P. Giudici, P. Green, and C. Tarantola. Efficient model determination for discrete graphical models. *Biometrika*, 2000. To appear.

[HCM$^+$00] D. Heckerman, D. Chickering, C. Meek, R. Rounthwaite, and C. Kadie. Dependency networks for density estimation, collaborative filtering, and data visualization. Technical Report MSR-TR-00-16, Microsoft Research, 2000.

[Hec89] D. Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proc. of the Conf. on Uncertainty in AI*, 1989.

[HT01] K. Humphreys and M. Titterington. Some examples of recursive variational approximations for Bayesian inference. In M. Opper and D. Saad, editors, *Advanced mean field methods*. MIT Press, 2001.

[IB98a] M. Isard and A. Blake. A mixed-state CONDENSATION tracker with automatic model-switching. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 1998.

[IB98b] M. Isard and A. Blake. A smoothing filter for condensation. In *Proc. European Conf. on Computer Vision*, volume 1, pages 767–781, 1998.

[Jaa01] T. Jaakkola. Tutorial on variational approximation methods. In M. Opper and D. Saad, editors, *Advanced mean field methods*. MIT Press, 2001.

[JGJS98] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. An introduction to variational methods for graphical models. In M. Jordan, editor, *Learning in Graphical Models*. MIT Press, 1998.

[JJ94a] F. V. Jensen and F. Jensen. Optimal junction trees. In *Proc. of the Conf. on Uncertainty in AI*, 1994.

[JJ94b] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[JKK95] C. S. Jensen, A. Kong, and U. Kjaerulff. Blocking-gibbs sampling in very large probabilistic expert systems. *Intl. J. Human-Computer Studies*, pages 647–666, 1995.

[JM00] D. Jurafsky and J. H. Martin. *Speech and language processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, 2000.

[Jor03] M. I. Jordan. An introduction to probabilistic graphical models, 2003. In preparation.

[JP95] R. Jirousek and S. Preucil. On the effective implementation of the iterative proportional fitting procedure. *Computational Statistics & Data Analysis*, 19:177–189, 1995.

[KFL01] F. Kschischang, B. Frey, and H-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans Info. Theory*, February 2001.

[Kja90] U. Kjaerulff. Triangulation of graphs – algorithms giving small total state space. Technical Report R-90-09, Dept. of Math. and Comp. Sci., Aalborg Univ., Denmark, 1990.

[KLC98] L. P. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[KLD01] K. Kask, J. Larrosa, and R. Dechter. Up and down mini-buckets: A scheme for approximating combinatorial optimization tasks. Technical Report R91, UC Irvine ICS, 2001.

[KP97] D. Koller and A. Pfeffer. Object-Oriented Bayesian Networks. In *Proc. of the Conf. on Uncertainty in AI*, 1997.

[KZ03] Vladimir Kolmogorov and Ramin Zabin. What energy functions can be minimized via graph cuts? *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 2003.

[Lau92] S. L. Lauritzen. Propagation of probabilities, means and variances in mixed graphical association models. *J. of the Am. Stat. Assoc.*, 87(420):1098–1108, December 1992.

[Lau95] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, 19:191–201, 1995.

[LBN96] B. Levy, A. Benveniste, and R. Nikoukhah. High-level primitives for recursive maximum likelihood estimation. *IEEE Trans. Automatic Control*, 41(8):1125–1145, 1996.

[LJ01] S. Lauritzen and F. Jensen. Stable local computation with conditional Gaussian distributions. *Statistics and Computing*, 11:191–203, 2001.

[LMP01] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Intl. Conf. on Machine Learning*, 2001.

[LN01] S. Lauritzen and D. Nilsson. Representing and solving decision problems with limited information. *Management Science*, 47:1238–1251, 2001.

[LP01] U. Lerner and R. Parr. Inference in hybrid networks: Theoretical limits and practical algorithms. In *Proc. of the Conf. on Uncertainty in AI*, 2001.

[LS98] V. Lepar and P. P. Shenoy. A Comparison of Lauritzen-Spiegelhalter, Hugin and Shenoy-Shafer Architectures for Computing Marginals of Probability Distributions. In G. Cooper and S. Moral, editors, *Proc. of the Conf. on Uncertainty in AI*, pages 328–337. Morgan Kaufmann, 1998.

[LT79] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math*, 36:177–189, 1979.

[Mac03] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[Min99] T. Minka. From Hidden Markov Models to Linear Dynamical Systems. Technical report, MIT, 1999.

[Min00] T. Minka. Bayesian linear regression. Technical report, MIT, 2000.

[Min01a] T. Minka. Algorithms for maximum-likelihood logistic regression. Technical report, CMU Statistics Tech Report 758, 2001.

[Min01b] T. Minka. Expectation propagation for approximate Bayesian inference. In *Proc. of the Conf. on Uncertainty in AI*, 2001.

[Min02] T. Minka. Bayesian inference in dynamic models: an overview. Technical report, CMU, 2002.

[MJ99] A. Madsen and F. Jensen. Lazy propagation: a junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence*, 113:203–245, 1999.

[MJ00] M. Meila and M. I. Jordan. Learning with mixtures of trees. *J. of Machine Learning Research*, 1:1–48, 2000.

[Mur98] K. P. Murphy. Fitting a conditional gaussian distribution. Technical report, U.C. Berkeley, Dept. Comp. Sci, 1998.

[Mur99] K. P. Murphy. A variational approximation for Bayesian networks with discrete and continuous latent variables. In *Proc. of the Conf. on Uncertainty in AI*, 1999.

[Mur02a] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, Dept. Computer Science, UC Berkeley, 2002.

[Mur02b] Kevin Murphy. Fast manipulation of multi-dimensional arrays in Matlab. Technical report, MIT AI Lab, 2002.

[Mur03] K. Murphy. Dynamic Bayesian Networks. In M. Jordan, editor, *Probabilistic Graphical Models*. 2003. To appear.

[MWJ99] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: an empirical study. In *Proc. of the Conf. on Uncertainty in AI*, 1999.

[Nea92] R. Neal. Connectionist learning of belief networks. *Artificial Intelligence*, 56:71–113, 1992.

[NH98] R. M. Neal and G. E. Hinton. A new view of the EM algorithm that justifies incremental and other variants. In M. Jordan, editor, *Learning in Graphical Models*. MIT Press, 1998.

[Nil98] D. Nilsson. An efficient algorithm for finding the M most probable configurations in a probabilistic expert system. *Statistics and Computing*, 8:159–173, 1998.

[NJ02] A. Y. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *NIPS-14*, 2002.

[Pas03] M. Paskin. Thin junction tree filters for simultaneous localization and mapping. In *Intl. Joint Conf. on AI*, 2003.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[Pea00] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge Univ. Press, 2000.

[PMM$^+$02] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *Advances in Neural Info. Proc. Systems*, 2002.

[Rab89] L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286, 1989.

[RD98] I. Rish and R. Dechter. On the impact of causal independence. Technical report, Dept. Information and Computer Science, UCI, 1998.

[RN02] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002. 2nd edition.

[RR01] A. Rao and K. Rose. Deterministically Annealed Design of Hidden Markov Model Speech Recognizers. *IEEE Trans. on Speech and Audio Proc.*, 9(2):111–126, February 2001.

[SDD90] R. Shachter, A. DelFavero, and B. D'Ambrosio. Symbolic probabilistic inference: a probabilistic perspective. In *AAAI*, 1990.

[SDW03] S. Sanghai, P. Domingos, and D. Weld. Dynamic probabilistic relational models. In *Intl. Joint Conf. on AI*, 2003.

[SGS00] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. MIT Press, 2000. 2nd edition.

[SHJ97] P. Smyth, D. Heckerman, and M. I. Jordan. Probabilistic independence networks for hidden Markov probability models. *Neural Computation*, 9(2):227–269, 1997.

[SJ02] H. Steck and T. Jaakkola. Unsupervised active learning in large domains. In *Proc. of the Conf. on Uncertainty in AI*, 2002.

[SK89] R. Shachter and C. R. Kenley. Gaussian influence diagrams. *Managment Science*, 35(5):527–550, 1989.

[SO01] D. Saad and M. Opper. *Advanced Mean Field Methods*. MIT Press, 2001.

[SP03] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In *Proc. HLT-NAACL*, 2003.

[SRG03] Ruslan Salakhutdinov, Sam T. Roweis, and Zoubin Ghahramani. Optimization with EM and Expectation-Conjugate-Gradient. In *Intl. Conf. on Machine Learning*, 2003.

[Ste00] M. Stephens. Dealing with label-switching in mixture models. *J. Royal Statistical Society, Series B*, 62:795–809, 2000.

[TAK02] B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Proc. of the Conf. on Uncertainty in AI*, 2002.

[TDW02] M. Takikawa, B. D'Ambrosio, and E. Wright. Real-time inference with large-scale temporal bayes nets. In *Proc. of the Conf. on Uncertainty in AI*, 2002.

[TK01] S. Tong and D. Koller. Active learning for structure in Bayesian networks. In *Intl. Joint Conf. on AI*, 2001.

[TMH99] Bo Thiesson, Christopher Meek, and David Heckerman. Accelerating EM for large databases. Technical Report MSR-TR-99-31, Microsoft Research, Redmond, WA, May 1999. Revised February 2001.

[TZ02] Z.W. Tu and S.C. Zhu. Image Segmentation by Data-Driven Markov Chain Monte Carlo. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24(5):657–673, 2002.

[UN98] N. Ueda and R. Nakano. Deterministic annealing EM algorithm. *Neural Networks*, 11:271–282, 1998.

[vdMDdFW00] R. van der Merwe, A. Doucet, N. de Freitas, and E. Wan. The unscented particle filter. In *NIPS-13*, 2000.

[WdM01] E. A. Wan and R. Van der Merwe. The Unscented Kalman Filter. In S. Haykin, editor, *Kalman Filtering and Neural Networks*. Wiley, 2001.

[Wei00] Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Computation*, 12:1–41, 2000.

[Wei01] Y. Weiss. Comparing the mean field method and belief propagation for approximate inference in MRFs. In Saad and Opper, editors, *Advanced Mean Field Methods*. MIT Press, 2001.

[WF99] Y. Weiss and W. T. Freeman. Correctness of belief propagation in Gaussian graphical models of arbitrary topology. In *NIPS-12*, 1999.

[WF01] Y. Weiss and W. T. Freeman. On the optimality of solutions of the max-product belief propagation algorithm in arbitrary graphs. *IEEE Trans. Information Theory, Special Issue on Codes on Graphs and Iterative Algorithms*, 47(2):723–735, 2001.

[WN01] E. A. Wan and A. T. Nelson. Dual EKF Methods. In S. Haykin, editor, *Kalman Filtering and Neural Networks*. Wiley, 2001.

[WSZ02] S. Wang, D. Schuurmans, and Y. Zhao. The latent maximum entropy principle. *IEEE Trans. on Information Theory*, 2002. Submitted.

[XJR03] E. P. Xing, M. I. Jordan, and S. Russell. A generalized mean field algorithm for variational inference in exponential families. In *Proc. of the Conf. on Uncertainty in AI*, 2003.

[XY01] Xiang and Ye. A simple method to evaluate influence diagrams. In *3rd Intl. Conf. Cognitive Science*, 2001.

[YFW01] J. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In *Intl. Joint Conf. on AI*, 2001.

[YFW02] J. Yedidia, W. Freeman, and Y. Weiss. Constructing free energy approximations and generalized belief propagation algorithms. Technical report, MERL, 2002.

[Zha98] N. Zhang. Probabilistic Inference in Influence Diagrams. *Computational Intelligence*, 14(4):475–497, 1998.

[ZP99] N. L. Zhang and D. Poole. On the role of context-specific independence in probabilistic reasoning. In *Intl. Joint Conf. on AI*, pages 1288–1293, 1999.

[ZP00] G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *Proc. Intl. Conf. Spoken Lang.*, 2000.

[ZY97] N. L. Zhang and Li Yan. Independence of causal influence and clique tree propagation. *Intl. J. of Approximate Reasoning*, 19:335–349, 1997.