Pless, D., and Luger, G., "EM Learning of Product Distributions in a First-Order Stochastic Logic Language." *Artificial Intelligence and Soft Computing: Proceedings of the IASTED International Conference*. IASTED/ACTA Press, 2003.

EM Learning of Product Distributions in a First-Order Stochastic Logic Language

Daniel Pless and George Luger
Department of Computer Science
University of New Mexico
{dpless, luger}@cs.unm.edu

Abstract

We describe a new logic-based stochastic modeling language called Loopy Logic. It is an extension of the Bayesian logic programming approach of Kersting and De Raedt [2000]. We specialize the Kersting and De Raedt formalism by suggesting that product distributions are an effective combining rule for Horn clause heads. We use a refinement of Pearl's loopy belief propagation [Pearl, 1998] for the inference algorithm. We also extend the Kersting and De Raedt language by adding learnable distributions. We propose a message passing algorithm based on Expectation Maximization [Dempster *et al.*, 1977] for estimating the learned parameters in the general case of models built in our system. We have also added some additional utilities to our logic language including second order unification and equality predicates.

1 Introduction

Several researchers [Kersting and De Raedt, 2000; Ngo and Haddawy, 1997; Ng and Subrahmanian, 1992] have proposed forms of first-order logic for the representation of probabilistic systems. In their paper "Bayesian Logic Programs", Kersting and De Raedt [2000] extract a particularly elegant kernel for developing probabilistic logic programs. They replace Horn clauses with conditional probability formulas. For example, instead of saying that `x` is implied by `y` and `z` (`x :- y, z`) they write that `x` is conditioned on `y` and `z` (`x | y, z`). They then annotate these conditional expressions with the appropriate probability distributions. In two valued logic, every symbol is either `true` or `false`. To support variables that can range over more than two values, they allow the domain of the logic to vary by predicate symbol. Kersting and De Raedt allow some predicates to range over other sets such as {`red`, `green`, `blue`}.

Ngo and Haddawy [1997] construct a logic-based language for describing probabilistic knowledge bases. Their knowledge database consists of a set of sentences giving a conditional probability distribution and a context under which this distribution holds. Such context rules do not appear in the language developed by Kersting and De Raedt [2000]. Both of these papers propose using Bayesian networks for inference. In our approach we construct Markov random fields for inference (Section 3).

Ng and Subramanian [1992] have a well developed formalism for probabilistic logic. Their system represents ranges of probabilities and provides rules for propagating these ranges through a probabilistic logic program. A simple restriction to a range of probability values is inherently non-Bayesian in nature. In a Bayesian framework, uncertainty in the value of a probability is handled through higher order probabilities. Ng and Subrahmanian's declarative language consists of sentences that contain horn clauses with terms that are annotated with probability ranges. The terms in the clauses are two valued. If the terms in the body are provably true, then the head is true with a probability bounded by the given range. Ng and Subrahmanian also show how to prove queries through PROLOG style SLD tree construction.

In a related approach, Friedman *et al.* [1999] develop a formalism based on the entity-relationship model that underlies most databases. This results in a logic that in some ways is more restrictive than that of Kersting and De Raedt [2000], but which allows second order aggregation functions.

Friedman *et al.* [1999] and Ngo and Haddawy [1997] can be viewed as extensions to the kernel extracted by Kersting and De Raedt [2000]. Our approach to probabilistic logic and inference further extends the language of Kersting and De Raedt by supporting product distributions and learning. Product distributions have been found to be an effective way of representing stochastic models for domains such as handwriting recognition [Mayraz and Hinton, 2000].

We have fully implemented our "Loopy Logic" probabilistic inference system. We have tested it in some standard domains such as Bayesian networks and Hidden Markov Models. Although the trials so far are on simple cases, we have evaluated the full functionality of the language including its ability to do parameter estimation or learning.

In the next section of this paper we describe our new language. In Section 3 we present inference through the construction of Markov fields and the use of loopy belief

propagation. In Section 4 we show how the same structure can be used for Expectation Maximization (EM) style parameter updates. In Sections 5 and 6 we demonstrate our language through examples of a Hidden Markov Model and digital circuit diagnosis. Finally, we present conclusions and future work in Section 7.

2 Language Description

We follow Kersting and De Raedt [2000] in the basic structure of our language. A sentence in the language is of the form `head | body1, body2, ..., bodyn = [p1, p2, ..., pm]`. The size of the conditional probability table (m) at the end of the sentence is equal to the arity (number of states) of the head times the product of the arities of the body. The probabilities are naturally indexed over the states of the head and the clauses in the body, but are shown with a single index for simplicity. For example, suppose x is a predicate that is valued over {red, green, blue} and y is boolean. P(x|y) is defined by the sentence `x | y = [[0.1, 0.2, 0.7], [0.3, 0.3, 0.4]]`, here shown with the structure over the states of x and y.

Terms (such as x and y) can be full predicates with structure and PROLOG style variables. For example, the sentence `a(X) = [0.5,0.5]` indicates that a is universally equally probable to take either of two values. The underline character (_) is used, as in PROLOG, to denote an anonymous variable. Also, as in PROLOG, the period is used for statement termination. We indicate the domain of terms with set notation. For example, `a ∈ {true, false}` indicates that a is either `true` or `false`. We include a shorthand in our language for singular distributions. To indicate that a variable has a deterministic value, for example, if a is `true`, then one can say `a = true` rather than `a = [1.0, 0.0]`. We also allow similar shorthand notation within larger structured distributions.

If we want a query to be able to unify with more than one rule head, some form of combining function is needed. Kersting and De Raedt [2000] allow for general combining functions. In our language, we restrict this combining function to one that is simple, useful, and works well with our inference algorithm. Our choice for combining sentences is a product distribution. For example, suppose we have two simple rules (facts) about some Boolean predicate a and one says that a is `true` with probability `0.4`, the other says it is `true` with probability `0.7`. The resulting probability for a is proportional to the product of the two. Thus a is `true` proportional to 0.4 * 0.7 and a is `false` proportional to 0.6 * 0.3. Normalizing, a is `true` with probability of about 0.61. Thus the overall distribution defined by a database in our language is the normalized product of the distributions defined for all the sentences.

One advantage of using the product rule for defining the resulting distribution is that observations and probabilistic rules are now handled uniformly. An observation is represented by a simple fact with a probability of 1.0 for the variable to take on the observed value. Thus a fact is simply a Horn clause with no body and a singular probability distribution, i.e., all the state probabilities are zero except for a single state.

We extend the basic structure of our probabilistic logic language in a number of ways. First, we allow second order terms, i.e., we can use variables for the function symbol in predicates. A useful example of this occurs with Boolean functions. If we have a group of predicates whose domain is `{true, false}` we can create a general `or` predicate:

```
or(X,Y) | X, Y =
    [[[1.0, 0.0], [1.0, 0.0]],
     [[1.0, 0.0], [0.0, 1.0]]].
```

Here `X` and `Y` in the body of the clause are higher order predicates. Now if we have two arbitrary predicates representing Boolean random variables, say `a(n)` and `b(m,q)`, we can form the predicate `or(a(n), b(m,q))` to get a random variable that is distributed according to the logical "or" of the two previous variables.

Our probabilistic logic language also supports simple Boolean equality predicates. These are denoted by angle brackets <>. For example, if the predicate `a(n)` is defined over the domain `{red, green, blue}` then `<a(n) = green>` is a variable over `{true, false}` with the obvious distribution. That is, the predicate is `true` with the same probability that `a(n)` is `green` and is `false` otherwise.

A further addition to our logic language is parameter fitting, i.e., learning. An example of a statement that indicates a learnable distribution is `a(X) = A`. The capital "A" indicates that the distribution for `a(X)` is to be fitted. The data for this is obtained from the facts and rules in the database itself. To specify an observation, one adds a fact to the database in which the variable `X` is bound. For example, suppose that we have the rule above and we add a set of five observations (the d_is) to give the following database:

```
a(X)  = A.
a(d1) = true.
a(d2) = false.
a(d3) = false.
a(d4) = true.
a(d5) = true.
```

In this case we have a single learnable distribution and five completely observed data points. The resulting distribution for a will be `true` 60% of the time and `false` 40% of the time. In this case the variables at each data point are completely determined. In general, this is not necessarily so, since there may be learnable distributions for which there are no direct observations. But a distribution can be inferred in the other cases and used to estimate the value of the adjustable parameter. In essence,

this provides the basis for an Expectation Maximization (EM) style algorithm [Dempster *et al.*, 1977] for simultaneously inferring distributions and estimating the learnable parameters (Section 4).

Learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Learnable distributions can also be parameterized with variables just as any other logic term. For example, the rule `rain(X, City) | season(X, City) = R(City)` indicates that the probability distribution for `rain` depends on the `season` and varies by `city`.

Similar to [Ngo and Haddawy, 1997], we support meta-predicates to allow the automated construction of rules. `rain(City) :- climate(City, C) = Rain(C)`, for example, indicates that the rain in a city is described by the climate for that city. So a non-probabilistic PROLOG term `climate(miami, tropical)` would indicate that the probability of rain in Miami, e.g., `rain(miami)`, is a learnable distribution (which is the same for all tropical cities).

All the elements described above have been implemented and tested. We are in the process of developing other predicate types, including probabilistic logic predicates. This is described in a preliminary form in the concluding section. We next describe the inference mechanism for the probabilistic language.

3 Inference

One of the simplest possible inference algorithms for Bayesian networks is the message passing algorithm known as *loopy belief propagation* first proposed by Pearl [1988]. This algorithm later had its effectiveness demonstrated by Murphy *et al.* [1999] after the connection between loopy belief propagation and Turbo Codes was pointed out [McEliece *et al.*, 1998]. In presenting our inference algorithm, we take an approach similar to Murphy *et al.* [1999] who represent stochastic models as Markov fields rather than Bayesian networks.

In Kersting and De Raedt's work, inference proceeds by constructing an SLD tree (a selective literal resolution system for definite clauses) and then converting it into a Bayesian network. We follow a similar path, but we convert the SLD tree to a Markov field instead. The advantage of our approach is that the product distributions that arise from goals that unify with multiple heads can be handled in a completely natural way. The basic idea is that random variable nodes are generated as goals are found. Cluster nodes are created as goals are unified with rules. In a logic program representing a Bayesian network, the head of a statement corresponds to a child node, while the clauses in the body correspond to the node's parents as seen in Figure 1a. To construct a Markov field, we add a cluster node between the child and its parents as is illustrated in Figure 1b. If more than one rule unifies with the rule head, then the variable node is connected to more than one cluster node, which results in a product distribution, as shown in Figure 2.

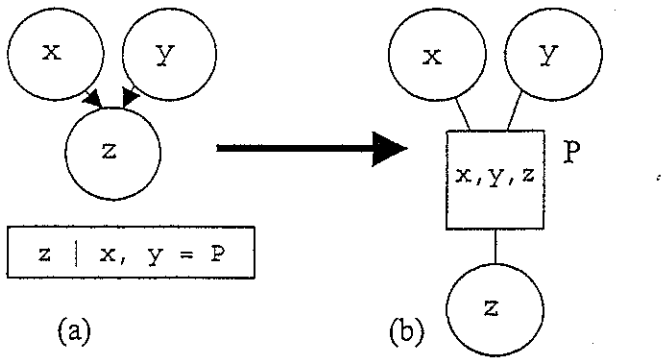

Figure 1: The transition of a piece of a Bayesian network into an equivalent piece of a Markov random field. Note that this generates a bipartite graph due to the addition of the cluster node, the square node which is annotated with the conditional probability distribution P.

As a result of the addition of the cluster nodes, the graphs that are generated for inference are bipartite as shown in Figure 1b. There are two kinds of nodes in these graphs, the variable and the cluster nodes. The variable nodes hold distributions for the random variables they define. The cluster nodes contain joint distributions over the variables to which they are linked. Messages between nodes are initially set randomly. On update, the message from variable node V to cluster node C is the normalized product of all the messages incoming to V other than the message from C. In the other direction, the message from a cluster node C to a variable node V is the product of the conditional probability table (the local potential) at C and all the messages to C except the message from V. This product is marginalized over the variable in V before being sent to V. This process, starting from random messages, and iterating until convergence, has been found to be effective for stochastic inference [Murphy *et al.*, 1999].

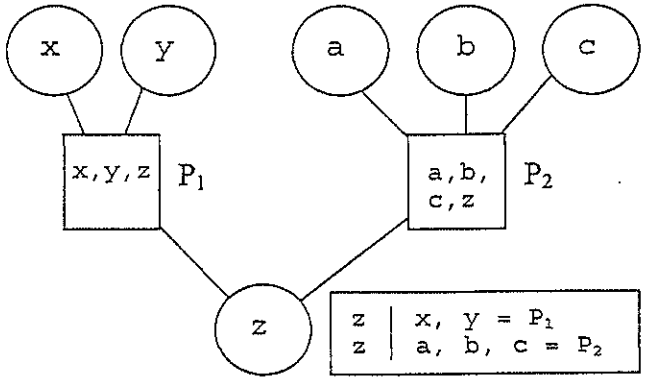

Figure 2: A product distribution is formed from two rules. This is represented in the Markov network as two cluster nodes attached to a single variable node.

The algorithm works by starting from a query (or possibly a set of queries) and generating the variable nodes that are needed. Each query is matched against all unifying heads in the database. The resulting bodies are then converted to new goals in the search. Our current system is limited in that goals produced by this search must be ground terms. Kersting and De Raedt [2000] place a range restriction on variables in terms: a variable may appear in the head of a rule only if it also appears in the body. As a result of this requirement, all facts entailed from the database are ground. By contrast, we require that all entailed goals be ground. We find that this requirement makes for better construction of useful models.

For the meta-predicates, the process proceeds similarly. When a rule head in a meta-predicate is matched, a standard PROLOG search is initiated on the body (between the `:-` and the `|` symbols) to instantiate their variables based on the pure PROLOG facts and rules in the database. All such bindings are then used to create the rules that are used in constructing the Markov random field as described above.

4 Learning

To support learning, we expand the process of building the Markov fields. When a cluster node is created that has a learnable distribution, a new learnable node is created (unless the appropriate node already exists). The parameter estimation example of Section 2, a small database based on the rule `a(X) = A`, is illustrated in Figure 3.

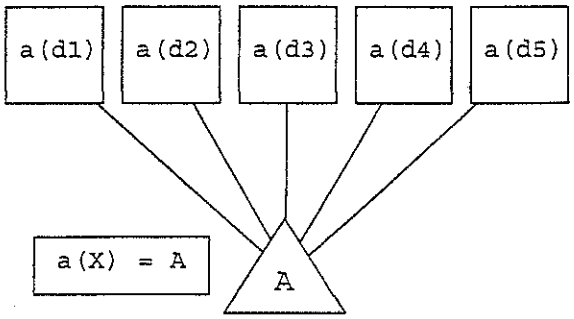

Figure 3: The learnable node and the associated cluster nodes that result from a learnable distribution with five datapoints.

We do parameter estimation with a message passing algorithm. Each learnable node is initially assigned a random normalized distribution. The conditional probability table is the learnable node's message to each of its linked cluster nodes. When the node is updated, each cluster node sends a message which is the product of all messages coming into that cluster. These (unnormalized) tables are an estimate of the joint probability at each cluster node. This is a distribution over all states of the conditioned and conditioning variables. The learnable node takes the sum of all these cluster messages. The result is then converted to a normalized conditional probability table.

By doing inference (*loopy belief propagation*) on the cluster and variable nodes, we compute the message for the learnable nodes. Applying the propagation algorithm until convergence yields an approximation of the expected values. This is equivalent to the Expectation step in the EM algorithm. The averaging that takes place over all the clusters gives a maximum likelihood estimate (the M step) of the parameters in a learnable node. Thus, allowing convergence in the variable and cluster nodes followed by updating the learnable nodes and iterating this process is equivalent to the full EM algorithm.

In the algorithm just described, we update all variables synchronously. This is not necessary and may not even be optimal. The nodes can be changed in any order, and updates of cluster and variable nodes may be overlapped with the updates of learning nodes. This iterative update process gives a family of EM style algorithms, some of which may be more efficient than standard EM [Dempster *et al.*, 1977] for certain domains. An algorithmic extension that this framework easily supports is the *generalized belief propagation* of Yedidia et al. [2000].

5 Example 1: A Hidden Markov Model

We next present an example showing how to construct a Hidden Markov Model (HMM) in our Bayesian logic. Suppose we have two states `{x, y}`. The system can start in either state, and at each time step, cycle to itself or transition to the other state. The probability of these events is a learnable distribution. In both states, the system can output one of two symbols `{a, b}`. The conditional distribution for these emissions is also represented by an adjustable distribution.

```
state ∈ {x,y}.
emit ∈ {a,b}.

state(s(N)) | state(N) = State.
emit(N) | state(N) = Emit.
```

The Hidden Markov Model works as follows. We represent each state with an integer, that is zero or the successor of another integer. We have implemented integer shorthand in our system, i.e., `2` is shorthand for `s(s(0))`. In the model, each state is conditioned on the previous state with the learnable distribution `State`. Each state emits its output with the learnable distribution `Emit`.

Strictly speaking, these four lines of code are sufficient to specify an HMM. We include the next five lines to demonstrate the utility of several of our other extensions, for example, the definition of the `and` predicate:

```
observed,o,and ∈ {true,false}.
```

```
and(X,Y) | X,Y =
    [true,false,false,false].

o([],N) = true.
o([H|T],N) =
    and(<emit(N)=H>,o(T,s(N))).

observed(L) = o(L,0).
```

Without these last five lines, one must specify an observed sequence by including in the database a separate fact for each emission that is seen. That is, one must state `emit(0) = a`, `emit(1) = b`, `emit(2) = b` and so on. With the additional five lines, three observations can be included with the predicate `observed([a,b,b])`.

We can easily express a product of HMMs by adding a new predicate to indicate the states of a second HMM. This new HMM is coupled to the existing one through a product distribution by using the same `emit` predicate. Here is an example of a second HMM with three states:

```
state2 ∈ {z,q,w}.
state2(s(N)) | state2(N) = State2.
emit(N) | state2(N) = Emit2.
```

Note that the final line uses the previous `emit` predicate which creates the product distribution. As a final comment, our language is far more general than is required to create simple HMMs.

6 Example 2: Diagnosing Digital Circuits

In a more extended example, we consider the diagnosis of combinatorial (acyclic) digital circuits. Assume there is a database of circuits that are constructed from *and*, *or*, and *not* gates and that we wish to model failures within such circuits. We assume that each component has a mode that describes whether or not it is working. The mode can have one of four values. The component is either good or has one of three failures: it is stuck with a value of one, stuck at zero, or intermittent, where the output of the element is random. We assume that the probability of the various failure modes is the same for components of the same type, although this probability may vary across types of components.

There are two questions that a probabilistic model can answer. First, assume the probabilities of failure are known. Given a circuit that isn't working properly, and one or more test cases (values for inputs and outputs), it would be useful to know the probability for each component mode in order to diagnose where the problem might be. The second question comes from relaxing the assumption that the failure probabilities are known. If there is a database of circuits and tests performed on those circuits, we may wish to derive from these tests what the failure probabilities might be.

We next provide code for this model. We use some conventions for naming variables. We let `Cid` be a unique ID for each circuit, and `T` be an ID for each different test, and `N` be an ID for a component of the circuit, and `Type` be the component type (*and*, *or*, *not*), and `I` be inputs (a list of `N`s) for the component.

The first two lines of the code are declarations to define which modes a component can be in as well as indicating that everything else is boolean:

```
val, and, or, not ∈ {v0, v1}.
mode ∈ {good, s0, s1, random}.
```

The `mode` and `val` statements provide the basic model for circuit diagnosis. The first indicates that the probability distribution for the mode of any component is a learnable distribution. One could put in a fixed distribution if the failure probabilities were known. Using the term `Mode(Type)` specifies that the probabilities may be different for different component types, but will be the same across different circuits. One could indicate that the distributions were the same for all components by using just `Mode` or that they differed across type and circuit by using `Mode(Type,Cid)`. The second statement of the two specifies how the possibility of failure interacts with normal operation of a component. The `val` predicate gives the output of component `N` in circuit `Cid` for test `Tid`.

```
mode(Cid, N) :- comp(Cid, N, Type, _) =
    Mode(Type).

val(Cid, Tid, N) :-
    comp(Cid, N, Type, I) |
    mode(Cid, N), Type(Cid, Tid, I) =
    [[v0,v1],[v0,v0],[v1,v1],
     [[0.5,0.5],[0.5,0.5]]].
```

The `and`, `or`, and `not` predicates model the random variables for what the output of a component would be if it is working correctly. The `and` and `or` are specified recursively. This allows arbitrary fan-in for both types of gates. The base case is handled by assigning a deterministic value for the empty list (one for `and`, zero for `or`). The recursive case computes the appropriate function for the value of the head of the list of inputs and then recurs. The `not` acts on a single value, inverting the value of the input component.

```
and(_, _, []) = v1.
and(Cid, Tid, [H|T]) |
    val(Cid, Tid, H), and(Cid, Tid, T) =
    [[v0,v0],[v0,v1]].

or(_, _, []) = v0.
or(Cid, Tid, [H|T]) = val(Cid, Tid, H),
or(Cid, Tid, T) =
    [[v0,v1],[v1,v1]].
```

```
not(Cid, Tid, N) | val(Cid, Tid, N) =
    [v1,v0].
```

Figure 4 presents an example circuit. The following four lines of code describe that circuit and the next three lines a sample test input case.

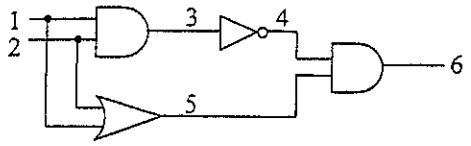

Figure 4: A sample circuit which implements xor.

```
comp(1, 3, and, [1,2]).
comp(1, 4, not, 3).
comp(1, 5, or, [1,2]).
comp(1, 6, and [4,5]).

val(1, 1, 1) = v0.   % input 1
val(1, 1, 2) = v1.   % input 2
val(1, 1, 6) = v0.   % output 6 - wrong
```

Without a powerful stochastic modeling tool, it is a non-trivial task to design a system that can diagnose digital circuit failures as well as estimate failure probabilities from a dataset of test cases. With our system, the basic model can be constructed using only nine statements. As the example shows, the representation of circuits and test data is transparent as well.

7 Conclusion

We have presented a new logic-based stochastic modeling language. We have applied a well-known effective inference algorithm, loopy belief propagation, to this language. This combination produces a first-order probabilistic language with the ability to represent product distributions effectively. We have also shown that learning is supported naturally within this framework.

In our view, each type of logic (deductive, abductive, and inductive) can be mapped to elements of our loopy logic language. The ability to represent rules and chains of rules is equivalent to deductive reasoning. Probabilistic inference, particularly from symptoms to causes, represents abductive reasoning. Finally, learning through the fitting of parameters to known datasets is a form of induction.

Some interesting extensions to our language are possible. For example, we plan to add continuous variables to our language. Using continuous variables it may be possible to support decision theory in the same framework. Furthermore, we would like to relax the constraint on rules that requires all goals be ground, thus producing a more expressive language. Finally, it may be interesting to allow the construction of the Markov field to be interleaved with the inference iterations, so that goals with an infinite SLD tree can be approximated.

Acknowledgements

The authors gratefully acknowledge the support of National Science Foundation under NSF grants IIS-9800929 and INT-9900485. The authors also thank Lance Williams, Terran Lane and Barak Pearlmutter for reviewing earlier drafts of this paper.

References

[Dempster *et al.*, 1977] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal statistical Society. Series B (Methodological)*, 39, 1, 1-38, 1977.

[Friedman *et al.*, 1999] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, 1999, 1300—1307, International Joint Committee on Artificial Intelligence.

[Kersting and De Raedt, 2000] K. Kersting and L. De Raedt. Bayesian logic programs. *AAAI-2000 Workshop on Learning Statistical Models from Relational Data, 2000. American Association for Artificial Intelligence.*

[Mayraz and Hinton, 2000] G. Mayraz and G. Hinton. Recognizing hand-written digits using hierarchical products of experts. *Advances in Neural Information Processing Systems 13*, 953-959, 2000.

[McEliece *et al.*, 1998] R. McEliece, D. MacKay, and J. Cheng. Turbo decoding as an instance of Pearl's 'belief propagation algorithm'. *IEEE Journal on Selected Areas in Communication*, 16(2), 140-152, 1998.

[Murphy *et al.*, 1999] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: an rmpirical study. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 467-475, San Francisco CA: Morgan Kaufmann, 1999.

[Ng and Subrahmanian, 1992] R. Ng and V. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150-201, 1992.

[Ngo and Haddawy, 1997] L. Ngo and P. Haddawy. Answering queries from context-sensitive knowledge bases. *Theoretical Computer Science*, 171:147-177, 1997.

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

[Yedidia *et al.*, 2000] J. Yedidia, W. Freeman, and Y. Weiss. Generalized belief propagation. *Advances in Neural Information Processing Systems, 13*, 689-695, Cambridge MA: MIT Press, 2002.