Readings #7: Approximate Inference*

Daphne Koller
Stanford University

Nir Friedman
Hebrew University

Exact inference algorithms exploit the structure of the distribution to make the inference significantly more efficient than simply summing out the joint. We've also seen how we can improve things even more by exploiting additional structure, e.g., the noisy-or structure. It is that which makes it possible for us to do inference in networks such as CPCS.

However, the algorithm is still limited, since it is still exponential in the largest factor formed, which may still be very large. Of course, this exponential blowup is unsurprising given that we proved that the basic problem is NP-hard. Perhaps if we make our problem a little easier we could do better. In this chapter, we consider what we can achieve if we were willing to give up on a little bit of accuracy in our results, i.e., content ourselves with an answer which is only approximately correct.

1 Overview

1.1 Evaluation metrics

Before we can discuss the problem of approximate inference, we need to have metrics for evaluating the quality of our approximation. We can consider two types of metric approximation: a local metric and a global metric. In the local metric, we are evaluating the quality of our answer to a particular query $P(\mathbf{x} \mid \mathbf{e})$. In the global metric, we are approximating the entire posterior $P(\cdot \mid \mathbf{e})$ using some other distribution $\hat{P}$, and evaluating the distance between P and $\hat{P}$.

Let us first consider local distance metrics. One obvious definition is:

Definition 1.1: Consider some real value ϵ and a query $P(\mathbf{x} \mid \mathbf{e})$. An estimate ρ has *absolute error* ϵ for $P(\mathbf{x} \mid \mathbf{e})$ if:

$$|P(\mathbf{x} \mid \mathbf{e}) - \rho| \in [1 - \epsilon, 1 + \epsilon] \blacksquare$$

This definition, although plausible, is somewhat weak. Consider, for example, a situation in which we are trying to compute the probability of a really rare disease, one whose true probability is, say, 0.00001. In this case, an absolute error of 0.0001 is unacceptable, even though such an error may be a wonderful approximation for an event whose probability is 0.3. A stronger definition would be:

Definition 1.2: The estimate ρ has *relative error* ϵ if

$$\frac{\rho}{P(\mathbf{x} \mid \mathbf{e})} \in [1 - \epsilon, 1 + \epsilon] \blacksquare$$

As we mentioned earlier, in some cases we wish to approximate the entire posterior distribution $P' = P(\cdot \mid \mathbf{e})$. Here also, there are some obvious metrics:

*This is based on draft of a text book, some of the references and citations are missing

Definition 1.3: Let μ and μ' be two distributions over the same probability space Ω. We define the L_1-*distance* between μ and μ' as:

$$\boldsymbol{D}_{L_1}(\mu;\mu') = \sum_{\omega\in\Omega} |\mu(\omega) - \mu'(\omega)|$$

We define the L_2-*distance* between μ and μ' as:

$$\boldsymbol{D}_{L_2}(\mu;\mu') = \left(\sum_{\omega\in\Omega} (\mu(\omega) - \mu'(\omega))^2\right)^{1/2}$$

∎

These distance metrics are general-purpose metrics used over any space. There is also a very useful distance measure which is defined for probability distributions.

Definition 1.4: Let μ and μ' be two distributions over the same probability space Ω. We define the *KL-distance* (for Kullback-Leibler distance) between μ and μ' as:

$$\boldsymbol{D}_{KL}(\mu;\mu') = \sum_{\omega\in\Omega} \mu(\omega)\log\frac{\mu(\omega)}{\mu'(\omega)}$$

This distance measure is also known as *relative entropy.* ∎

We note several things about this definition. First, it is asymmetrical: μ plays a very different role from μ'. Second, we were very careful to call it a *measure* as opposed to a metric. A distance metric $\boldsymbol{D}(;)$ satisfies certain properties:

positivity: $\boldsymbol{D}(\mu;\mu')$ is always non-negative, and is zero if and only if $\mu = \mu'$;

symmetry: $\boldsymbol{D}(\mu;\mu') = \boldsymbol{D}(\mu';\mu)$;

triangle inequality: for any three distributions μ, μ', μ'', we have that

$$\boldsymbol{D}(\mu;\mu') \leq \boldsymbol{D}(\mu;\mu'') + \boldsymbol{D}(\mu'';\mu')$$

KL-distance satisfies only the first of these three properties.

Nevertheless, KL-distance is a very natural distance measure for distributions, especially in asymmetric setting where μ is the "true" distribution, and μ' is the approxiomation. There are many motivations for it, some of which we will encounter when we discuss the problem of learning from data. For the moment, consider the following information-theoretic justification.

[[information-theoretic justification]]

KL-distance also have certain very important properties. For example:

Proposition 1.5: *Let μ and μ' be joint probability distributions over a set of random variables $\mathbf{X}\cup\mathbf{Y}$ (for $\mathbf{X}$ and $\mathbf{Y}$ disjoint sets). Then*

$$\boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X},\mathbf{Y});\mu'(\mathbf{X},\mathbf{Y})) = \boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X});\mu'(\mathbf{X})) + \sum_{\mathbf{x}} \mu(\mathbf{x})\boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{Y}\mid\mathbf{x});\mu'(\mathbf{Y}\mid\mathbf{x}))$$

where $\mu(\mathbf{X})$ denotes the marginal distribution over $\mathbf{X}$, and $\mu(\mathbf{Y}\mid\mathbf{x})$ denotes the conditional distribution over $\mathbf{Y}$ given a particular value $\mathbf{x}$ for $\mathbf{X}$.

In other words, KL-distance decomposes nicely for compound joint distributions. There are several useful corollaries to this result:

Corollary 1.6: *If $I(\mathbf{X};\mathbf{Y})$ holds in both μ and μ', then*

$$\boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X},\mathbf{Y});\mu'(\mathbf{X},\mathbf{Y})) = \boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X});\mu'(\mathbf{X})) + \boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{Y});\mu'(\mathbf{Y})).$$

Corollary 1.7: *For any $\mu(\mathbf{X},\mathbf{Y})$ and $\mu'(\mathbf{X},\mathbf{Y})$,*

$$\boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X});\mu'(\mathbf{X})) \leq \boldsymbol{D}_{\mathrm{KL}}(\mu(\mathbf{X},\mathbf{Y});\mu'(\mathbf{X},\mathbf{Y}))$$

This last result is particularly useful, as it allows us to convert global bounds into local bounds. In fact, we also know that KL-distance can be used to bound L_1-distance:

Theorem 1.8: *For any two distribution μ and μ', we have that*

$$\boldsymbol{D}_{L_1}(\mu;\mu') \leq \left(2\ln 2\boldsymbol{D}_{\mathrm{KL}}(\mu;\mu')\right)^{1/2}.$$

Thus, if we have a global bound on KL-distance, we can convert it to a local bound on KL-distance for the marginal over any query variables using Corollary 1.7, and then into a local L_1 bound (which implies an absolute error bound) using Theorem 1.8.

1.2 Complexity analysis

With these definitions, we can turn to answering the question of whether approximate inference is actually an easier problem. For many years, based on the hope that approximation is, indeed, much easier, people developed a variety of algorithms for approximate probabilistic inference. It was only in 1993, with a paper by Dagum & Luby, that people realized the bad news. Approximation is also NP-hard! This result is fairly easy to see for the case of relative errors.

Theorem 1.9: *The problem of finding ρ which has a relative error ϵ to $P(X = x)$ in a BN is NP-hard.*

Proof: The proof is obvious based on the original NP-hardness proof for exact Bayesian network inference (Theorem **??**). Then, we proved that it is NP-hard to decide whether $P_B(X) > 0$. Now, assume that we have an algorithm that returns an estimate ρ to the same $P_B(X)$ which is guaranteed to have relative error ϵ for some $\epsilon < 1$. Then $\rho > 0$ iff $P_B(X) > 0$. Thus, achieving this relative error is as NP-hard as the original problem. ∎

What about absolute error? In this case, as we will soon see, the problem of just approximating $P(X = x)$ has a randomized polynomial time algorithm, and therefore cannot be NP-hard unless NP is in RP.[1]. This is an improvement, because even the problem of computing $P(X = x)$ is hard in the exact case. Unfortunately, it is NP-hard to find an absolute approximation to $P(x \mid \mathbf{e})$ for any $\epsilon < 1/2$. As $\epsilon = 1/2$ corresponds to random guessing, this result is quite discouraging.

Theorem 1.10: *Let ϵ be any number in $(0, 1/2)$. The problem of finding ρ which has absolute error ϵ to $P(x \mid \mathbf{e})$ is NP-hard.*

[1] The complexity class RP is the class of problems solvable by randomized algorithms in polynomial time. Common belief is that NP $\not\subseteq$ RP

Proof: The proof uses the same construction that we used before. Consider a formula ϕ, and consider the analogous BN $\mathcal{B}$, as described in Theorem **??**. Recall that our BN had a variable Q_i for each propositional variable q_i in our boolean formula, a bunch of other intermediate variables, and then a variable X whose value, given any assignment of values ${q_1}^1, {q_1}^0$ to the Q_i's, was the associated truth value of the formula. We now show that, given such an approximation algorithm, we can decide whether the formula is satisfiable. We begin by computing $P(Q_1 \mid x^1)$. We pick the value v_1 for Q_1 which is most likely given x^1, and instantiate it to this value. That is, we generate a network $\mathcal{B}_2$ which does not contain Q_1, and which represents the distribution $\mathcal{B}$ conditioned on $Q_1 = v_1$. We repeat this process for $Q_2, \ldots, Q_n$. This results in some assignment $v_1, \ldots, v_n$ to the Q_i's. We now prove that this is a satisfying assignment iff the original formula ϕ was satisfiable.

First, the easy case. If ϕ is not satisfiable, then $v_1, \ldots, v_n$ can hardly be a satisfying assignment for it. Now, assume that ϕ is satisfiable. We show that it also has a satisfying assignment with $Q_1 = v_1$. If ϕ is satisfiable with both $Q_1 = {q_1}^1$ and $Q_1 = {q_1}^0$, then this is obvious. Assume, however, that ϕ is satisfiable, but not when $Q_1 = v$. Then necessarily, we will have that $P(Q_1 = v \mid x^1)$ is 0, and the probability of the complementary event is 1. If we have an approximation ρ whose error is guaranteed to be $< 1/2$, then choosing the v which maximizes this probability is guaranteed to pick the v whose probability is 1. Thus, in either case the formula has a satisfying assignment where $Q_1 = v$. We can continue in this fashion, proving by induction on k that ϕ has a satisfying assignment with $Q_1 = v_1, \ldots, Q_k = v_k$. Thus, in the case where ϕ is satisfiable, this process will terminate with a satisfying assignment. In the case where ϕ is not, it clearly will not. We can determine which is the case simply by checking whether the resulting assignment satisfies ϕ. This gives us a polynomial time process for deciding satisfiability. ∎

1.3 Overview of techniques

Despite these pessimistic results, the algorithms developed can be quite useful in many situations. They seem to work fairly well in practice, and some even come with guaranteed performance bounds for certain important subclasses of networks (as we will see).

There is a wide variety of very different approximation algorithms. Here is a list of the major categories:

- Instantiation-based methods: approximate the joint by focusing on the more likely entries.
- Random sampling: generate random samples and use them as an approximation. The two main subtypes here are:
  - Forward sampling, which generates samples forward through the network
  - Markov Chain Monte Carlo, which gradually converges to generating samples from the posterior
- Structural approximation: simplify the network. Again, there are many different types:
  - Partial evaluation: focus on a subnetwork by ignoring less relevant nodes; this is the technique used in the CPCS network.
  - Structural simplification: eliminate weak dependencies, abstract values, abstract variables (e.g., by collapsing multiple fault nodes).
  - Variational methods: an extremely successful and very recent method; construct a simple network (one with fewer edges) that minimizes the KL-distance to the exact posterior.

  – Approximation during inference: in variable elimination or clique tree inference, simplify the intermediate factors to allow easier propagation. This is the basis for inference in conditional linear Gaussian hybrid networks, and for a new algorithm for reasoning in dynamic Bayesian networks.

- Loopy belief propagation: pass messages as if the network is singly connected (a polytree) ignoring the cycles. Surprisingly, this process often converges to a good answer. It is the basis for the most successful ever coding algorithm for high-bandwidth communication over noisy channels.

2 Instantiation-based methods

One important class of approximation algorithm is based on the following intuition. We are interested in summing out the joint distribution. Unfortunately, the joint distribution is very large, and we cannot afford to explore all of it. Instantiation-based methods approximate the joint using some of its entries $\mathbf{x}[1], \ldots, \mathbf{x}[N]$. As we will see in the next section, random sampling methods use the same idea, but generate random samples from the joint. By contrast *deterministic search methods* approximate the joint distribution by focusing on the most likely entries in it. If we enumerate those entries that count for a lot of mass, we may be able to approximate various probabilities.

Most naively, we can enumerate various different full assignments to all variables $\mathbf{x}[1], \ldots, \mathbf{x}[N]$ and compute the probability for each one. We then estimate

$$P(q \mid \mathbf{e}) \approx \frac{\sum_{i=1}^{N} \eta(q, \mathbf{e} \mid \mathbf{x}[i]) P(\mathbf{x}[i])}{\sum_i \eta(\mathbf{e} \mid \mathbf{x}[i]) P(\mathbf{x}[i])}$$

where $\eta(\mathbf{y} \mid \mathbf{x}[i])$ is an *indicator variable* which takes the value 1 if $\mathbf{y}$ holds in the instantiation $\mathbf{x}[i]$ and 0 otherwise.

Of course, we don't want to pick just an arbitrary set of N assignments. We want to pick likely ones. Unfortunately, it is not so easy to pick the most likely assignments. In fact, this problem is also NP-hard. Even worse, even if we could somehow find the assignments which were more likely a priori, that wouldn't be enough. In most cases, our most likely assignments are inconsistent with our evidence, and would simply be thrown out. For example, in the Alarm example, the most likely scenario is that there is neither a burglary or an earthquake, there is no alarm, no phone call, and no radio report. However, the model is most interesting when one of these events did occur. This situation is also common in medical diagnosis, where the model is not interesting in the high-probability scenario that the person is completely healthy.

Thus, one type of improvement to a basic search algorithm involves generating instantiations in a way that tries to make them consistent with the evidence.

Another improvement involves the fact that we don't always need to generate full entries in the joint. It is often enough to generate samples that give values only to some of the variables. You'll see one example in your problem set.

As another example, recall that exact inference in singly connected networks can be done in linear time. Therefore, we can use an algorithm that only instantiates enough nodes to make the network singly connected. Consider the network in Figure 1. If we instantiate A to some particular value, we have eliminated the dependence between the two chains. In each instantiation, the induced network is singly connected, and exact inference is easy. We then simply traverse all possible instantiations, do a computation in each one, and aggregate the results by summing them up, weighted by the probabilities of the different instantiations.

This operation is called *cutting the loop*. It leads to a general algorithm called *cutset conditioning*, which essentially conditions on enough variables to cut all loops, rendering the network singly connected. Full cutset conditioning is an exact inference algorithm, which can be viewed as doing the summation of the joint "from the outside in". That is, letting X_1 be the root variable, we sum out the joint by considering all values x_1 of X_1, and doing a separate computation for each of them. If we continue doing the computation from the outside in, we would end up with the standard exponential algorithm that generates the entire joint distribution. In cutset conditioning, we stop when the network induced over the variables remaining in the summation is singly connected, and then do the dynamic programming / variable elimination algorithm that works from the inside-out. We note that, in terms of the operations performed in generic variable elimination, there are no computational benefits to doing cutset conditioning; we can only lose by failing to exploit the dynamic programming. However, it does allow us to trade off space for time. In the diamond network, we can do a separate computation for each x_1^i, and simply accumulate the results. We would never need to generate factors that are larger than a single CPD. If we were to use dynamic programming, we would need to maintain factors over pairs of variables, requiring more space.

We can turn cutset conditioning into an approximation algorithm by not considering all instantiations of the cutset variables. Instead, we focus on certain instantiations that have high probability. By summing out only those, we obtain an approximate inference algorithm called *bounded conditioning*.

Search based algorithms provide little to none a priori performance guarantees. However, as you'll see in the problem set, they often come with "online" performance bounds. That is, as the algorithm runs, it provides upper and lower bound estimates on the probability, allowing you to stop when the error gets small enough.

3 Random Sampling

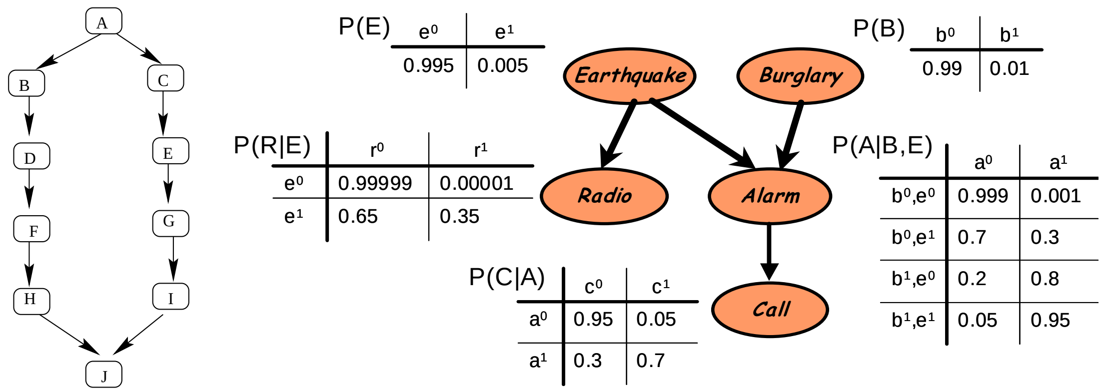
The problem with picking instantiations deterministically is that, in the worst case, our choice may be arbitrarily bad. If we pick instantiations at random, then at least we hedge our bets. Thus, we randomly sample instantiations until we get a reasonable estimate. Like the search-based algorithms, we generate a set of samples $\mathbf{x}[1], \ldots, \mathbf{x}[N]$, and use them to estimate the probabilities of various events.

It turns out that randomized algorithms are often much easier to analyze than deterministic ones. Thus, in this case, we actually have algorithms that, for a certain interesting class of networks, allow us to guarantee a certain relative error with high probability (i.e., assuming that our random coin tosses are not consistently off). Partially for this reason, stochastic simulation algorithms are very popular in practice. Another reason is that they are very easy to implement.

3.1 Rejection sampling

The simplest random sampling algorithm is called *rejection sampling*. It simply generates samples from the prior distribution defined by the network. This is a very simple process. To understand it, consider the Alarm network, shown again in Figure 1.

We begin by sampling B with the appropriate (unconditional) distribution; i.e., we "toss a coin" that lands "heads" 1% of the time and tails the remaining 99%. Say that it landed "heads", so that we pick the value b^1 for B. Similarly, we sample E from its distribution; say that the result is e^0. Given those, we know the right distribution from which to sample A: $P(A \mid b^1, e^0)$, as defined



(a) (b)

Figure 1: (a) A multi-connected network; (b) The Alarm network.

by A's CPD; we therefore pick A to be a^1 with probability 0.8 and a^0 with probability 0.2. The process continues similarly for R and C.

In general, we sample the nodes in some order consistent with the partial order of the BN, so that by the time we sample a node we have values for all of its parents. The sample is clearly from the distribution defined by the CPD and by the chosen values for the node's parents.

Given a bunch of samples $\mathbf{x}[1], \ldots, \mathbf{x}[N]$, we can estimate the probability of any event. For any event $\mathbf{y}$, we estimate

$$\hat{P}(\mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} \eta(\mathbf{y} \mid \mathbf{x}[i])$$

For example, our estimate for the probability of an event such as b^1, c^1 is the average number of samples in which B got the value b^1 and C the value c^1. Note that, when we were using deterministically generated samples, we multiplied the ith term in the summation by $P(\mathbf{x}[i])$. Why do we not do that here? Because we are already sampling $\mathbf{x}[i]$ with that probability. If $\mathbf{x}[i]$ has higher probability, we will sample it more often. If we also multiplied by it, we would be double counting its weight.

This sampling process requires linear time in the size of the network to produce each sample. The overall cost is, in turn, linear in the number of samples. So the only issue is the number of samples required for a good estimate.

First, let's understand this at an intuitive level. Assume we are trying to estimate $P(q \mid \mathbf{e})$. Then we generate a sample from the network using the process above. Note that samples that are incompatible with $\mathbf{e}$ are useless: they participate neither in the computation of $\hat{P}(q, \mathbf{e})$ nor in the computation of $\hat{P}(\mathbf{e})$. In other words, all samples incompatible with our evidence are rejected; we estimate the probability $P(q \mid \mathbf{e})$ by counting the fraction *within the nonrejected samples* that are consistent with q.

Thia observation reveals the limitation of this approach. If our evidence is very unlikely, then most of the samples we generate will be rejected. As a result, our estimate for $P(q \mid \mathbf{e})$ will be based on only a small number of samples, resulting in a poor approximation. Alternatively, we would have to continue, generating many samples, until we get enough for a reasonable estimate.

Theorem 3.1: *Let ρ_S be the estimate for $\hat{P}(\mathbf{y})$ resulting from a randomly chosen sample set S with N samples. Then in order to guarantee that ρ_S has relative error at most ϵ with probability at leat $1 - \delta$, it is enough to have $N \geq (4/P(\mathbf{y})\epsilon^2) \cdot \ln(2/\delta)$.*

This theorem tells us how many samples we need. We can get a good relative error to $P(q \mid \mathbf{e})$ using a reasonable number of samples if both $P(\mathbf{e})$ and $P(q, \mathbf{e})$ are not too small. (Because we can get a good relative estimate for each one.) To get a good absolute error approximation for $P(q \mid \mathbf{e})$ it is enough for $P(\mathbf{e})$ to be not too small: if $P(q, \mathbf{e})$ is small, then even a large relative error leads to a small absolute error.

3.2 Likelihood weighting

Looking at the logic sampling process, it seems very wasteful. We produce entire samples that we know are going to be useless, since they are incompatible with my evidence. It seems much more intuitive to simply force the samples to take on the appropriate values at observed nodes. That is, when we come to sampling a node X_i whose value has been observed, we simply set it to its observed value.

Of course, we can't simply do that. In order to preserve the correctness of the algorithm, we have to account for the fact that X_i would only have taken this value in some of the samples. How do we do that? Consider our example, and assume that we have evidence c^1. Now, assume that we are in the middle of a sampling process, and have sampled e^1, b^0, a^0. We want to force C to take the value c^1. However, if we just go ahead and do that, we are ignoring the fact that C is actually unlikely to take this value if a^0. We need to consider the probability that, had we done the regular sampling process, we would have ended up with c^1. This probability is only 0.3. Thus, this sample where we force C to take this value should only be worth 30% of a full sample. What if we also observed r_0? Given e^1, we would have sampled r^1 with probability 0.65. We would have sampled both c^1 and r^1 with probability $0.3 \cdot 0.65$.

More generally, we have the following algorithm:

LikelihoodWeighting($\mathbf{e}$)
  for $j = 1, \ldots, N$
    $w[j] = 1$
    for $i = 1, \ldots, n$
      Let $\mathbf{u}$ be the assignment to $\mathbf{Pa}_{\mathbf{X_i}}$ in $\mathbf{x}[j]$
      If X_i hasn't been observed
        Sample $\mathbf{x}[j]_i$ from $P(X_i \mid \mathbf{Pa}_{\mathbf{X_i}} = \mathbf{u})$
      If X_i has been observed to be x_i
        Set $\mathbf{x}[j]_i = x_i$
        Set $w[j] = w[j] \cdot P(x_i \mid \mathbf{Pa}_{\mathbf{X_i}} = \mathbf{u})$

This algorithm is called *Likelihood Weighting*, because the weights correspond roughly to the likelihood of the evidence. The algorithm is doing a correct computation, in the following sense:

Theorem 3.2: *Let $p(\mathbf{x})$ be the probability that the algorithm generates a particular instantiation $\mathbf{x}$ as our sample. Let $w(\mathbf{x})$ be the weight of that sample, as computed by the algorithm. Then $p(\mathbf{x}) \cdot w(\mathbf{x})$ is precisely $P(\mathbf{x}, \mathbf{e})$. I.e., it is $P(\mathbf{x})$ for $\mathbf{x}$ consistent with $\mathbf{e}$ and 0 otherwise.*

Therefore, we have that for any instantiation $\mathbf{x}$ consistent with $\mathbf{e}$,

$$\frac{1}{N} \sum_i \eta(\mathbf{x} \mid \mathbf{x}[i]) w(\mathbf{x}[i])$$

converges to $P(\mathbf{x})$ for large N. However, only instantiations $\mathbf{x}$ consistent with my evidence $\mathbf{e}$ are generated. Thus, for any event $\mathbf{y}$, we have that

$$\begin{aligned} P(\mathbf{y}, \mathbf{e}) &= \sum_{\mathbf{x} \text{ consistentwith } \mathbf{y}} P(\mathbf{x}, \mathbf{e}) \\ &= \sum_{\mathbf{x}} \eta(\mathbf{y} \mid \mathbf{x}) P(\mathbf{x}, \mathbf{e}) \\ &\approx \sum_{\mathbf{x}} \eta(\mathbf{y} \mid \mathbf{x}) \frac{1}{N} \sum_i \eta(\mathbf{x} \mid \mathbf{x}[i]) w(\mathbf{x}[i]) \\ &= \frac{1}{N} \sum_i \eta(\mathbf{y} \mid \mathbf{x}[i]) w(\mathbf{x}[i]) \end{aligned}$$

In particular, we can estimate $P(\mathbf{e})$ by

$$\sum_i \eta(\mathbf{e} \mid \mathbf{x}[i]) w(\mathbf{x}[i]) = \sum_i w(\mathbf{x}[i]).$$

Thus, to estimate $P(q \mid \mathbf{e})$ we simply compute

$$\frac{\sum_j w[j] \eta(q \mid \mathbf{x}[j])}{\sum_j w[j]}.$$

As the number of samples grows, both numerator and denominator converge to the right number, and therefore so does our estimate.

To understand the convergence properties of this process better, let's abstract away from the details. We are interested in a posterior distribution (given our evidence). In the best of all possible worlds, we would generate samples from that joint posterior distribution, thereby getting a good estimate of the posterior probability of various events. However, we don't have the posterior, so we can't generate samples from it. Thus, our sampling process generates samples from a different distribution, and uses the weights in order to compensate for the difference. The quality of our approximation depends largely on how close we are to sampling from the posterior.

Let's look at two cases. Assume that all of the evidence in our network is at the roots. In this case, we are sampling from the posterior already, and all samples will have weight 1. What if all of the evidence is at the leaves? In this case, we are sampling purely from the prior distribution, leaving the correction purely to the weights. This works OK if the prior is similar to the posterior. To understand when this can be the case, consider the fact that the prior is a weighted average of the posteriors, weighted over different instantiations of the evidence:

$$P(\mathbf{X}) = \sum_{\mathbf{e}} P(\mathbf{e}) P(\mathbf{X} \mid \mathbf{e})$$

If the evidence is very likely, then it is a major component in this summation, and it cannot be too far from the prior. For example, the posterior distribution in the Alarm network given the evidence r^0 is very similar to the prior. However, for unlikely evidence, the weight of $P(\mathbf{X} \mid \mathbf{e})$ is negligible, and there is nothing constraining the posterior to be similar to the prior. Indeed, our distribution given r^1 is very different from our prior. In this case, our sampling process is going to generate a lot of samples where r^1 is very unlikely; these samples are not going to be very useful relative to the correct posterior distribution $P(\mathbf{X} \mid r^1)$.

We note that likelihood weighting is a special case of a more general approach called *Importance Sampling*. In importance sampling, we sample from one distribution μ' as a substitute for sampling from μ. We weight the samples to compensate for the difference between μ' and μ. (See the homework for details.) In general, the closer μ' is to μ, the better our sampling process. We can use this more general framework to improve the algorithm. For example, we can try to move our sampling distribution closer to the posterior. Of course, the problem is that we don't know the posterior. However, as we sample more and more, we eventually get a better picture of what our correct posterior distribution looks like. So, one possibility is, as we get more samples, to start sampling from a distribution which is closer to what our previous samples tell us is the correct one. In other situations, we can use this approach to target our sampling process differently, e.g., to focus more attention (samples) on areas in the distribution that we feel are important (e.g., ones associated with particularly malignant diseases).

3.3 Markov Chain Monte Carlo sampling

Likelihood weighting makes better use of our samples by forcing them to match our evidence. If evidence is in the middle of the network, it also makes our sampling distribution closer to the posterior distribution, by taking into consideration at least part of the evidence (the upstream evidence) when generating the samples. An alternative approach is to take the evidence into consideration throughout the process. One algorithm that does that is *Gibbs sampling*, an instance of a general class of algorithms called *Markov Chain Monte Carlo methods.* This is a very important technique, with many many applications.

The basic idea is the following: we generate a "random walk" over assignments that are consistent with the evidence. The distribution with instantiations are encountered in our random walk will converge to the true posterior distribution. We begin by instantiating all of the observed nodes to their observed values, and the other nodes to some randomly selected values. Then, we iteratively go over the unobserved variables, and pick a new value for each of them one by one.

GibbsSampling
For each observed variable X_i, set x_i to its observed value
For each unobserved variable X_i, set x_i to a random value
Repeat
  Pick some unobserved variable X_i
  Sample x_i from $P(X_i \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$
Until done

Fortunately, $P(X_i \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ is easy to compute (see homework).

This process is guaranteed to eventually produce samples from the posterior distribution. However, it leaves many unanswered questions. First, the initial samples are atypical, so it may take a while before we get a real sample from the posterior distribution. Thus, we have to run things for a while before we can start using the samples (a stage called *burn-in*). Second, having gotten one such sample, subsequent samples are not independent of it. That can skew our sampling process to one part of the distribution space unless we are careful. In general, there is quite a bit of black magic in getting Gibbs sampling to behave well. However, the algorithm is fundamental and can deal with situations which likelihood weighting is incapable of handling.

4 Structural Approximation

A different class of techniques is based on the observation that the cost of exact inference depends directly on the complexity of the network: the sizes of the CPDs, the number of long-range edges, etc. Therefore, we can simplify the network a little, and then do exact inference on the result. There are several approaches:

- Drop edges.
- Drop less relevant nodes.
- Modify network structure in general, e.g., collapse several nodes into one, as in the single fault hypothesis transformation that we saw in the child care network.
- Abstract a variable by aggregating some of its values; this idea is particularly useful in the context of discretization.
- Simplify the CPDs of a network in order to put it in an easier class.

Here also, evidence is an issue. If our knowledge engineering process put in these nodes, edges, or values for a node, then they are probably there for a reason. I.e., there are circumstances in which they matter. If we just drop them arbitrarily, then our network may be very wrong in some circumstances. Thus, in order to do this process well, we must target our approximation to the circumstances, i.e., the query and the evidence.

Let's consider, as an example, the problem of abstracting the value space of a node. Assume we have a node *Cholesterol*, with values *low*, *marginal*, and *high*, which we denote c_0, c_1, c_2, and that we want to abstract the two values *marginal* and *high* into the single value *unhealthy*. We denote this new value of the node c'. The first question is the specification of the abstracted network, specifically, the CPDs involving C. The CPD of C is easy. Letting $\mathbf{U}$ be the parents of C, we simply define

$$P(c' \mid \mathbf{u}) = P(c_1 \mid \mathbf{u}) + P(c_2 \mid \mathbf{u}).$$

What about the CPD of a variable T that depends on C as well as on some other set of parents $\mathbf{Y}$? What we really want is the weighted average of the two CPD distributions:

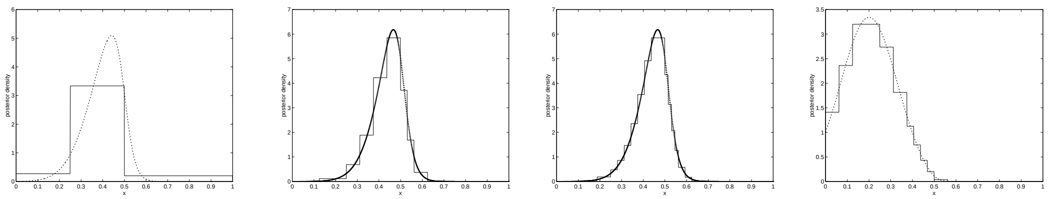
$$P(T \mid c', \mathbf{y}) = \frac{P(T \mid c_1, \mathbf{y})P(c_1 \mid \mathbf{y}) + P(T \mid c_2, \mathbf{y})P(c_2 \mid \mathbf{y})}{P(c_1 \mid \mathbf{y}) + P(c_2 \mid \mathbf{y})}.$$

However, that requires that we do exact inference at least once in the network, thereby defeating the process. Therefore, we often simply use an unweighted average.

Is this transformation "correct"? I.e., if we work in the transformed network, and do not care about C, are our results exact? No, because our conditional independencies may not hold for the abstracted value space.

A special case of this problem is the discretization of a continuous variable, e.g., location. Intuitively, if we believe that the object is very unlikely to be in some part of the space, it's probably OK to make that entire part a single region. However, the part where we think the object might be probably needs to be represented more accurately.

Of course, we are back to a cyclic question, as before. In order to figure out where the object probably is, we need to do inference. But we need this information before we do the inference. One trick, which actually works well in practice, is to do an initial phase of inference with a network which is a very rough approximation, and then to use that as guidance in order to refine the network in the important places. Figure 2 shows an example in the context of discretization.



Initial discretization Phase I Phase III Different evidence

Figure 2: Example of discretization that takes evidence into account.

5 Loopy belief propagation

The last method that we describe was invented by Pearl and then largely abandoned, because it seemed so clear that it was flawed. Roughly speaking, this method does inference in the network as if there are no loops, ignoring the correlations arising from multiple parallel paths of influence.

To understand the algorithm, let us revisit exact inference in the context of a loop-free network, i.e., a polytree. In this case, we can view our clique-tree propagation algorithm as having a cluster $\mathbf{C}_X$ for each node X, containing X and its family. The cluster of a node X is connected to the clusters of its parents and the clusters its children.[2] Now, consider the standard message passing algorithm applied to this cluster tree. To send a message to one of its neighbors, a node collects the messages from all of its neighbors (parents and children), multiplies them with its CPD, sums out all variables except the desired recipient, and sends out the message. To make the algorithm more uniform, we will view evidence as a message. That is, for each node X, we will introduce a "message to itself" $\mu_{X\rightarrow X}(X)$. If our evidence includes $X = x'$, we will have that

$$\mu_{X\rightarrow X}(x) = \begin{cases} 1 & x = x' \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

If we have no evidence on X, then $\mu_{X\rightarrow X}(X) \equiv 1$. We will also define $\pi_0[X]$ to be X's CPD: $P(X \mid \mathbf{Pa_X})$.

Now, consider a node X, and let $U_1, \ldots, U_k$ be its neighbors. The message $\mu_{X\rightarrow U_i}()$ that X passes to its neighbor U_i is the product:

$$\mu_{X\rightarrow U_i}(\mathbf{C}_X \cap \mathbf{C}_{U_i}) = \sum_{\mathbf{C}_X - \mathbf{C}_{U_i}} \mu_{X\rightarrow X}(X)\pi_0[X] \prod_{\ell \neq i} \mu_{U_\ell \rightarrow X}(\mathbf{C}_{U_\ell} \cap \mathbf{C}_X) \tag{2}$$

In other words, each cluster executes the standard clique tree message passing operation: to pass a message to cluster i, it takes the messages from its other neighboring clusters, multiplies them with each other and with its own CPD, reduces the factor to one consistent with the evidence, and then sums out all variables except the ones shared with cluster i, and sends that as the resulting message.

Note that, as the network has no loops, the intersection between any pair of neighboring clusters is a single variable; hence, the messages are over single variables. However, the messages between $\mathbf{C}_X$ and $\mathbf{C}_{U_i}$ are different when U_i is a parent of X and when it is a child. In the first case, $\mathbf{C}_X$ contains U_i, and the message between the two clusters is a message over U_i. In the second case,

[2] In a real clique tree, we might not have a clique for each node. If Y is a node whose only child is X, then the family of Y will be a subset of the family of X, and $\mathbf{C}_X$ will subsume $\mathbf{C}_Y$. However, it is useful to view the algorithm in terms of a cluster for each node, as it makes for a more uniform presentation.

$\mathbf{C}_{U_i}$ contains X, and the message is over X. Otherwise, however, the message passing operation is identical.

When applying this algorithm to a polytree, the leaves and roots of the network can send out their messages immediately. Then, the nodes that are distance 1 from the fringe can send out their messages, and so on. Eventually, all messages will be sent, and the algorithm will terminate. The node can compute its own marginal probability by doing

$$belX = \alpha \sum_{X} \mu_{X \to X}(X) \sum_{\mathbf{Y}} \prod_{j} \mu_{Y_j \to X}(X) \sum_{\mathbf{U}} \prod_{\ell} \mu_{U_\ell \to X}(X) \qquad (3)$$

where α is a normalizing constant. For singly connected networks, $belX = P(X \mid \mathbf{e})$, where $\mathbf{e}$ is our evidence.

We now apply this algorithm to the case where the networks contain loops. In this case, we do not generate a cluster *tree*. Rather, we have a cluster *graph*: Each node X is associated with a cluster $\mathbf{C}_X$ which contains X and its parents. Each cluster $\mathbf{C}_X$ is directly connected to $\mathbf{C}_U$ for each neighbor (parent or child) U of X. In this case, clearly the graph induced over clusters is not a tree.

In this case, we no longer necessarily have "leaves" in the tree that can start the message passing process without waiting for a message from anyone else. Hence, we consider a "parallel" version of this algorithm. We start by initializing all the messages to be uniformly vectors of 1's. Then, in parallel, all of the cliques recompute all of the messages using Eq. (3), and substitute the old values with the new values. Note that all of the messages are (re)computed simultaneously, before any of them change. This step is called an *iteration* of belief propagation.

It is important to note several aspects of this algorithm. First, unlike the case of polytrees, the intersection between neighboring clusters can have more than one variable. For example, if X is a parent of Y, but they have a shared parent Z, then the intersection between $\mathbf{C}_X$ and $\mathbf{C}_Y$ will contain both X and Z. As written, Eq. (3) tells us that the messages between the clusters should be over all the variables in the intersection. Nevertheless, the standard implementation of belief propagation uses the same messages as in the singly-connected case. In this case, the message between $\mathbf{C}_X$ and $\mathbf{C}_Y$, for Y a parent of X, will contain only X. We note, however, that recent results suggest that an algorithm that passes messages over all variables in the intersection has somewhat better properties [**?**].

A second note relates to the fact that the messages sent keep shrinking in every iteration. In each iteration, we are multiplying the incoming messages with the CPD, and then zeroing those entries that are inconsistent with the evidence. As we iterate this process, we will get serious difficulties with numerical stability. (This issue was not critical in the case of clique tree propagation, because messages were not passed indefinitely, but only once up and down the tree.) Hence, it is customary to renormalize messages before they are sent out, thereby preventing this phenomenon. Note that renormalizing a message does not change the overall result of the algorithm. When a message is renormalized, all of its entries are multiplied by the same constant factor. That same constant factor is propagated to other messages computed using this one; i.e., all entries of subsequent messages will also be multiplied by this constant factor. Ultimately, when the final beliefs are computed, this constant factor simply disappears in the final normalization process.

Overall, the algorithm is shown in Figure 3.

Is this algorithm doing the right thing? If the network is singly-connected (a polytree), we know that the cluster graph is a tree. It is easy to show that, in this case, the algorithm converges in a finite number of steps. (See exercise.) However, if the graph is not singly-connected (loopy), the situation is more complex. Consider the simple diamond in Figure 1, and assume we have evidence

Procedure Loopy-Belief-Propagation(
    Graph over $X_1, \ldots, X_n$, // Bayesian network structure
    $P(X_i \mid \mathbf{Pa_{X_i}})$, // CPDs for Bayesian network nodes
    $u_1, \ldots, u_m$ // evidence
)
    For each $i = 1, \ldots, n$
        Let $\mathbf{C}_i$ be $X_i \cup \mathbf{Pa_{X_i}}$
        Define $\mu_{i \to i}(X_i)$ as in Eq. (1)
        Define $\pi_0[i]$ to be $P(X_i \mid \mathbf{Pa_{X_i}})$
        For each j such that $X_i \to X_j$
            Set $S_{i,j}$ and $S_{j,i}$ to be $\{X_j\}$
        Define $\mathcal{N}_i$ to be the parents and children of X_i
    For each i and $j \in \mathcal{N}_i$
        Set $\mu_{i \to j}(S_{i,j})$ to be uniformly 1
    Iterate
        For each i and $j \in \mathcal{N}_i$
            Set $\tau_{i \to j}(S_{i,j})$ to be $\sum_{\mathbf{C}_i - S_{i,j}} \mu_{i \to i}(X_i) \pi_0[i](\mathbf{C}_i) \prod_{\ell \in \mathcal{N}_i, \ell \neq j} \mu_{\ell \to i}(S_{\ell,i})$
        For each i and $j \in \mathcal{N}_i$
            Set $\mu_{i \to j}(S_{i,j}) := \tau_{i \to j}(S_{i,j})$
    For each i
        Set $bel(X_i) := \alpha \sum_{\mathbf{C}_i - i} \mu_{i \to i}(X_i) \pi_0[i](\mathbf{C}_i) \prod_{\ell \in \mathcal{N}_i} \mu_{\ell \to i}(S_{\ell,i})$
            where α is a normalizing constant

Figure 3: Loopy belief propagation

about node D. In this algorithm, node A double counts the evidence about J, which influences it via two different paths. We also lose the convergence of the algorithm: if node B sends a message to node A, it can travel around the loop, and ultimately influence $\mu_{D \to B}(B)$; as a result, D will have to revise its message to A, causing the entire process to repeat. Indeed, Pearl himself said [**?**, p.195]:

> When loops are present, the network is no longer singly connected and local propagation schemes will invariably run into trouble ... If we ignore the existence of loops and permit the nodes to continue communicating with each other as if the network were singly connected, messages may circulate indefinitely around the loops and the process may not converge to a stable equilibrium ... Such oscillations do not normally occur in probabilistic networks ... which tend to bring all messages to some stable equilibrium as time goes on. However, this asymptotic equilibrium is not coherent, in the sense that it does not represent the posterior probabilities of all nodes of the networks.

Pearl's observation is true. It is possible to construct examples where this process does not converge.

This approach was largely abandoned for many years, until a group of engineers invented a new approach for coding and decoding messages [**?**]. This approach is a new solution to the fundamental problem of sending a message over a very noisy channel, and recovering it from the garbled result. This problem was analyzed theoretically by Shannon [**?**], who showed that there were inherent limitations on the amount of information that we can send through such a channel, bounds that depend both on innate bandwidth of the channel and on the signal to noise ratio. His analysis, however, was not constructive, and he did not propose a code that came even close to achieving these innate limits.

Since his work, various groups gradually came up with better and better codes, but none of them came even close to the Shannon limit until the work of Berrou *et al.* [**?**]. They came up with a new and fairly simple coding scheme that, empirically, came much closer to achieving the Shannon limit than any other code proposed so far! The problem was that their decoding algorithm had no theoretical justification, and, while it seemed to work well in real examples, could be made to diverge or even converge to the wrong answer.

The basic algorithm they proposed was as follows. Consider the task of sending a message consisting of n bits $U_1, \ldots, U_n$. In a simple coding scheme, we might send each U_i, but in addition intersperse the U_i's with additional bits that encode some transformation of the U_i's. For example, we might define a set of new bits to be the parity of some subset of the original message bits (in a way that allows the reconstruction of the original message). The resulting message is $X_1^a, X_1^b, \ldots, X_n^a, X_n^b$, where the a denotes an original bit and the b a transformed bit. The actual received message is some garbled version of these $2n$ bits $Y_1^a, Y_1^b, \ldots, Y_n^a, Y_n^b$. Now, if we knew $X_1^a, \ldots, X_n^a$, we would have the message. If we knew $X_1^b, \ldots, X_n^b$, we could also reconstruct the message exactly. However, all we have is a noisy version of these bits. However, assume we have a decoder D^a that can take the noisy bits $Y_1^a, \ldots, Y_n^a$ and tell me a probability over the value of each real bit X_i^a. This decoder can (and should) take into consideration how likely each value for the bit was in the first place. But given a prior probability for the value of each bit, it returns a posterior probability based on the noisy bits $Y_1^a, \ldots, Y_n^a$. Similarly, we have a decoder D^b that does the same for the bits $Y_1^b, \ldots, Y_n^b$.

The idea of Berrou *et al.* was to "turbo-charge" the decoders by using a feedback loop. We would use D^a to decode $\mathbf{Y}^a$, getting a probability over each X_i^a. We would then convert that to a probability over each X_i^b, and use that prior and D^b to decode $\mathbf{Y}^b$ and get a new set of probabilities over the bits X_i^b. We then convert this to a set of probabilities over the X_i^a, and repeat the whole process with the $\mathbf{Y}^a$ again. This process continues until it converges.

This process looked a lot like black magic, and in the beginning, many people did not even believe them that the algorithm worked. However, they tried the process themselves, and eventually, this was hailed as "the most exciting and potentially important development in coding theory in many years". At this point, there is widespread agreement in the coding community that these codes, called *Turbo Codes*, "represent a genuine, and perhaps historic, breakthrough."

However, it took a while after these codes were accepted before anyone (including the inventors) realized that the decoding algorithm was actually performing belief propagation on a loopy Bayesian network! We can view this process as a Bayesian network, with a random variable for each bit involved. A Bayesian network representing a simple convolutional code with a memoryless channel (i.e., one that makes independent errors on each bit, as opposed to longer bursts) is shown in Figure 4. The structure of the turbocode network is similar but more complex; it also allows for burst noise on the channel, in which case the variables representing received bits would not be conditionally independent given the variables representing the bits sent.

If we could do exact inference on this network, we could figure out the most likely message given the observed bits. This would be the optimal decoding scheme. However, in complex codes, the transformed bits are a complex transformation of several message bits, leading to a network with complex connectivity and large cliques. This makes exact algorithms infeasible, particularly given the real-time constraints of decoding messages.

It turns out that loopy belief propagation works very well on these networks. In most practical cases, it converges to the most likely codeword. More precisely, the belief values that it converges to

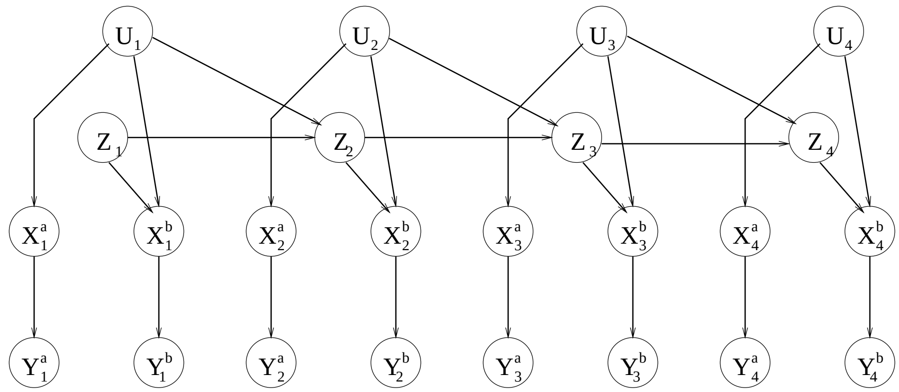


Figure 4: A Bayesian network for a simple convolutional code with a memoryless channel and a four bit message.

are wrong, but the most likely value is usually correct. This observation has created a lot of interest in loopy belief propagation, and there has been a lot of work on analyzing both loopy propagation and its connection to coding. This has had several effects. From a practical perspective, people have developed other coding schemes that use the same idea, and are much better even than the original turbo-codes.

From a more general perspective, there has been a lot of theoretical and empirical work on loopy belief propagation in general Bayesian networks. Experiments show that, although there are cases where the algorithm does not converge, it often does. And when it does, it often gives astonishingly good answers. People have also started to analyze the theoretical properties of loopy belief propagation on general Bayesian networks, and to generalize the algorithm. This work is ongoing, and there are many new results. One interesting set of results holds for two very different classes of networks: networks with only binary variables and a single loop, and continuous networks with linear Gaussian dependence. There is an interesting similarity in the results for these two very different classes of networks:

- In the binary networks, the mode (most likely value) of each variable is guaranteed to be correct, but the probabilities are not; in Gaussian networks, the means are guaranteed to be correct but the variances are not.
- In both networks, the rate of convergence is correlated with the quality of the approximation of the ultimate probabilities.

This similarity of results in two such very different classes of networks leads one to hope that this type of analysis also holds in a broader class of networks. There are also more recent results for other classes of networks; this is a very active research area, with new results every few months.

There has also been work on extending the algorithm. For example, Weiss *et al.* have shown that the loopy belief iteration process works even better if one considers clusters that are not just the families of nodes. One can use larger clusters, and even messages over more than one variable (if the "cluster separators" — the intersection between neighboring clusters — is larger than a single variable). This approach gives even better results, empirically, than the "base" loopy belief propagation.