

# Pearl's Belief Propagation Algorithm and Loopy Bayesian Networks

Casey Boardman  
cdb@ccs.neu.edu

April 21, 2004

## Abstract

In this paper we review the idea of “loopy” belief propagation. Specifically, we will look into the possibility of using Pearl’s belief propagation algorithm on Bayesian networks containing undirected cycles. It is known that this algorithm will give incorrect numerical results in this type of graph, and possibly incorrect maximum a posteriori probability (MAP) estimates. It is also known that these problems are, for general Bayesian networks, NP-Hard [6]. This use of belief propagation was inspired by the recognition that the turbo code decoding algorithm (and its success) is in essence an instance of belief propagation on a loopy network.

## 1 Summary

This paper is broken up into sections. In section 2 we will briefly review Bayesian networks. In section 3 we review Pearl’s Belief Propagation algorithm for polytrees. Section

4 will review turbo codes, and section 5 will discuss their relation to belief propagation. Section 6 will discuss other situations where belief propagation works and why, as well as where it may not. In section 7 we will analyze the run-time complexity of Bayesian networks in general.

## 2 Review of Bayesian Networks

A Bayesian network is a directed acyclic graph (DAG) made up of nodes and causal edges. Each node has a probability of having a certain value. For our purposes all nodes will be binary, though a Bayesian network may have  $n$ -ary nodes. We define a parent and child nodes as follows: a directed edge exists from a parent to a child. Each child node will have a conditional probability table based in parental values. There are no directed cycles in the graph, though there may be “loops”, or undirected cycles. An example

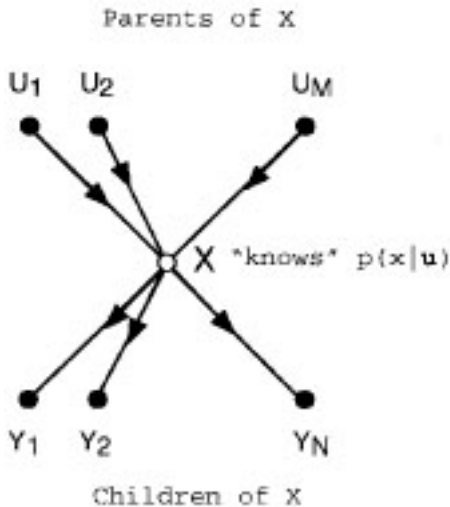


Figure 1: A polytree network. [9]

network is shown in 1, with parents  $U_i$  sharing a child,  $X$ .  $X$  is a child of the  $U_i$ 's as well as being a parent to the  $Y_i$ 's.

### 3 Pearl's Polytree Algorithm

Pearl's algorithm is similar to the classic "forward-backward" algorithm for information sharing. The forward-backward algorithm can be described using the example of a line of soldiers. Let's say the head soldier wishes to know how many soldiers are in the line. One naive way of doing this would be to ask each of his neighbors (in this case the one behind him) for how many soldier there are, counting themselves, to the end of the line. This steps proceed recursively, to the

end of the line, when the end soldier replies "one", the second from the end adds one to this and replies "two", and so on. One problem with this is that if a new soldier adds on to the end of the line, the head soldier will not know until he re-queries.

A different algorithm is have each soldier report the number of soldiers there are behind them (adding on themselves, of course) and re-reporting this is it changes. For example, in the initial case, the end soldier will report "one" the next "two" and so on, and the head soldier will get the correct count. If a soldier adds on the back of the line, he will say "one". The soldier that had formerly reported one will now report "two", and on up the line, so that the head soldiers knowledge of the number of soldier in the line will change as the line length changes.

Pearl's algorithm works similarly for polytrees, or networks containing no loops. Let us consider a given node  $X$  having parents  $U_1 \dots U_m$  and children  $Y_1 \dots Y_n$ , as shown in figure 1. Evidence will be represented with  $E$ , with evidence "above"  $X$  (evidence at ancestors of  $X$ ) written as  $e^+$  and evidence "below"  $X$  (evidence at descendants of  $X$ ) written as  $e^-$ .

Knowledge of evidence can flow both down the network (from parent to child) or up the network (child to parent). Let us label these messages as  $\pi_{U_i}$  and  $\lambda_{Y_j}$  messages, respectively. What we wish to compute is the belief of  $X$  having the value of true or false ( $BEL(X = true)/BEL(X = false)$ ). We will write this as  $BEL(X)$ , with the value written as the normalized vector (true false). For example, if we know that  $X$  is true,

$BEL(X) = (1 \ 0)$ . The belief of  $X$  given it's parent and child values is

$$BEL(X) = P(X|e) = \alpha \lambda(X) \pi(X) \quad (1)$$

where  $\alpha$  is a normalizing value,  $\lambda(X) = P(e^-|x)$ , and  $\pi(X) = P(x|e^+)$ .

$$\lambda(X) = \prod_{i=1}^j \lambda_{Y_i} \quad (2)$$

$$\pi(X) = \sum_{\vec{u}} P(x|\vec{u}) \pi(\vec{u}) \quad (3)$$

$\vec{u}$  is a vector of possible values for  $U_1 \dots U_m$ .

Once  $BEL(X)$  has been updated,  $X$  must send  $\pi$  and  $\lambda$  messages to it's children and parents, respectively. But we must ensure not to update  $Y_i$ 's belief based on evidence from  $Y_i$ , and likewise for  $U_i$ . (For example, if John tells me it is supposed to rain today, my belief that it will rain should increase. But I should not increase John's belief that it will rain based on the fact that he told that he thinks it will rain. His belief should only go up if I also have reasons to think it will rain from other sources.) This is done by calculating the messages to send to  $Y_i$  and  $U_i$  as follows:

$$\pi(Y_i) = \frac{BEL(X)}{\lambda_{Y_i}} \quad (4)$$

and

$$\lambda(U_i) = \beta \sum_X \lambda(X) \sum_{u_k: k \neq i} P(X|\vec{u}) \prod_{k \neq i} \pi(u_k) \quad (5)$$

where  $\beta$  is a normalizing factor, and  $\vec{u}$  is the vector of values for  $U_1 \dots U_m$ .

The details of these formulae are not as important to our discussion as is the understanding of the message-passing idea of the algorithm. Essentially, each node tells it's neighbors what it believes, based on prior probability and the belief of it's neighbors. For a more in depth review of Pearl's algorithm, please refer to Chapter Four of [8].

To summarize, the basic steps involved are these:

1. initialize the network
2. update beliefs
3. propagate changes in belief
4. if beliefs changed, loop on step 2

## 4 Turbo Codes

Turbo codes were introduced by Berrou, Glavieux, and Thitimajshima in the early 1993 [2]. The strength of turbo codes is that they come close to the Shannon limit, or maximum capacity, in bits per second, of a communications channel for a give power level [5]. The basic process of sending a bit string with turbo encoding is as follows [1]:

1. Generate an encoding of the bit string using a systematic encoder, ENC1
2. Run the bits string through an interleaver, generating a scrambled version for the original string (this is a key step)

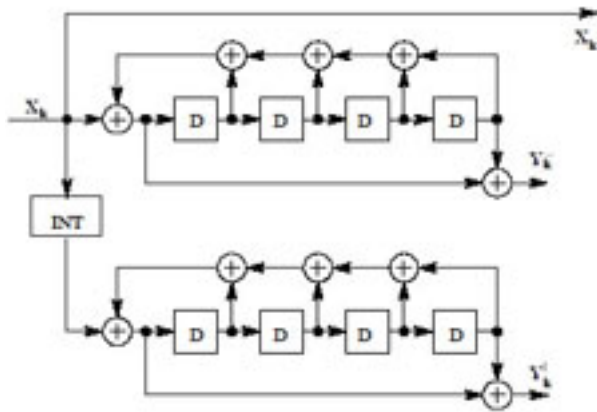


Figure 2: A rate half turbo code. (D represents a flip-flop) [1]

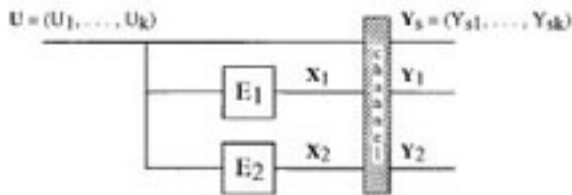


Figure 3: The basic setup for turbo encoding. [9]

3. Generate an encoding of the scrambled string using an encoder, ENC2, similar to step 1.
4. Send the string along with the parity bit strings output from ENC1 and ENC2

The received string is then decoded, with the probability of each bit received assigned based on the signal. For an example of this “soft-input” without parity, let us assign a

signal of +1 to be a bit of 1, and a signal of -1 to be 0. If we then receive a message of (-1, 0, 1), we would initially believe the sent bit string to be  $0x1$ , with  $x$  having a 50% chance of being 0 or 1. If we received, (-.9, .2, .8) we would guess a bit string of 011.

The turbo decoding algorithm is as follows. Two decoders are constructed, DEC1 and DEC2. The input to decoder 1 is the transmitted bit string and the parity bits from ENC1. The input to DEC2 is the transmitted bit string and the parity bits from ENC2. All transmitted bits are assigned a probability of 1/0 as described above. The two decoders then compute their belief of what the original bit string was, with the original bits each having an even chance of being 1 or 0. Their beliefs are then shared, and each decoder recomputes their belief and the process is repeated a set number of times, typically four to ten.

The final belief of the two decoders is then added and converted back to binary (in our example, all positive values go to 1, all negative to 0).

One drawback of turbo codes is the time it takes to decode, so they aren’t an option for uses such as voice transmission or disk storage, but in areas where delay is acceptable (such as picture, video, and mail transmission). They are also being used in current space communications, such as the European Space Agency’s SMART-1 [5].

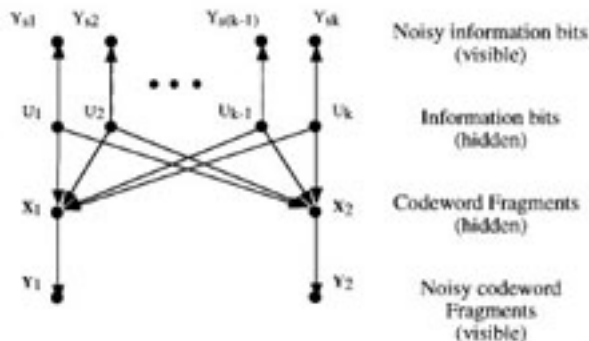


Figure 4: Turbo decoding as a Bayesian network. [9]

## 5 Turbo decoding as an instance of Pearl’s algorithm

In this section we describe how the turbo-decoding algorithm is an instance of Pearl’s belief propagation algorithm. Observe that the turbo decoder can be constructed as a Bayesian network according to 4, where  $U = u_1, u_2, \dots, u_n$  represents the original bit string.  $X_1$  and  $X_2$  represent the encoded strings created by passing  $U$  through the encoder ENC1 and ENC2.  $Y_{s1}, Y_{s2}, \dots, Y_{sn}$  represents the transmitted bits of  $U$ ,  $Y_1$  the transmitted bits of  $X_1$ , and  $Y_2$  the transmitted bits of  $X_2$ . Upon receiving a transmission, we have a prior probability on all  $u_i$  of  $(.5 \ .5)$ , given probabilities for the bits of  $X_1$  and  $X_2$  based on the encoding algorithm, and a given belief of  $Y_s, Y_1$ , and  $Y_2$  based on the signal received. Please note that the belief propagation algorithm is not guaranteed

to give the correct answer, or even converge, due to the loops in the network. (One example loop is  $U_1 \rightarrow X_1 \rightarrow U_2 \rightarrow X_2 \rightarrow U_1$ .)

Once we have entered the evidence (from the signal), by the belief propagation algorithm, we propagate evidence one step, updating our beliefs for the bits of  $X_1, X_2$ , and  $U$ . We now propagate another step. Note that in this step evidence from  $U$  influences belief in  $X_1, X_2$ , and  $Y_s$ , as well as evidence from  $X_1$  and  $X_2$  influencing  $U$ . Beliefs are updated according to the new messages. Once again, we propagate another step. At this point, evidence from  $Y_2$  that influenced beliefs at  $U$  will cause changes in belief for  $X_1$ , thus DEC2 “shares belief” with DEC1. This process is then repeated until a set number of times, or until the beliefs converge.

Though there are cases for which Pearl’s algorithm will fail to converge for turbo codes, though it has shown to be quite useful in real-world applications [13]. This information has caused interesting research into the subject of why Pearl’s algorithm seems to work in this case, and in what other areas it can be applied.

## 6 Belief Propagation on loopy networks

Let’s begin our analysis with a look at a network with a single loop. Given evidence somewhere on the loop, the evidence will be double-counted. In other words, the beliefs of node will propagate around the loop in both directions. This will obviously give us incor-

rect belief for any given node on the loop. But it has been shown that for graphs with a single loop[12]:

1. Unless all the compatibilities are deterministic, loopy belief propagation will converge.
2. An analytic expression relates the correct marginals to the loopy marginals.
3. If the hidden nodes are binary, then the loopy beliefs and the true beliefs are both maximized by the same assignments, although the confidence that the assignments is wrong for the loopy beliefs.

In our turbo code example, however, we do not have just a single loop, but multiple loops. Though there have been examples where this algorithm has converged on an incorrect answer [10], in empirical studies it has been shown to converge on the correct answer a high percentage of the time.

These statements were cursorily verified by the program written by me to analyze and understand Pearl’s algorithm, and what results one gets with differing types of numerical and structural data. For more information on the program, please see A.

It has also been shown that if the beliefs do not converge, there is little that can be done to compute the actual value as the oscillations have very little correlation to the correct marginals [7].

The big question, then, is “when does using loopy propagation make sense?”. It has

been empirically shown that this approach works well in the area of error-correcting codes (such as turbo codes) as well as in computer vision. One property that the Bayesian networks for these problems have in common (that I have noticed) is the “shallowness” of the network, with each network having many loops. (An example for a vision network is the PYRAMID [7] network). Much like the turbo code network, many nodes are on loops, and the loops themselves are small. I hypothesize that this parallelism of the loops helps to keep beliefs from becoming too exaggerated, as they may if the loops are longer, or if there are a series of loops.

There has also been recent studies showing that is a connection between Pearl’s algorithm and certain approximations of variational free energy in statistical physics[6]. The analysis of belief propagation as related to that subject is beyond the scope of this paper, but this information is included in the case someone with an interest in this subject reviews this paper.

## 7 Analysis of Bayesian Networks

It was proven in 1990 [3] that exact computation of conditional probabilities for a given Bayesian network is NP-hard. Analysis shows the runtime for the brute force method of enumeration is  $O(q^m)$ , where  $q$  is the size of the alphabet (for a binary network as we have studied  $q = 2$ ) and  $m$  is the number of unknown variables. We have seen that Pearl’s

algorithm, for the special case of a polytree, has an efficient runtime of  $O(Nq^e)$ , where  $e$  is the maximum number of parents on a vertex [9]. It can be seen that in that in the case of the turbo-decoding algorithm, the runtime is linear in the size of the network, as evidence is propagated a constant number of times. It was shown in 1993 that approximating the probabilistic inference is also intractable in the worst case [4]. In many cases with Bayesian networks, however, what we are really interested in is not the actual numerical distributions of the conditional probabilities, but the MAP estimate. This, however, was also shown to be NP-hard in the worst case by Shimony in 1994 [11]. We will briefly present this proof here.

The problem we are given is, for evidence  $E$ , we wish to find a value assignment  $A$  over all variables  $V$  so as to maximize  $P(A|E)$ . The related decision problem is to decide if there exists an assignment for  $A$  so that  $P(A|E) \geq p$ , for a given  $p$ . We will prove that this decision problem is NP-complete, and therefore the maximization problem is NP-hard.

We will construct a Bayesian network  $BN = (V, E, P)$  consisting of the vertex set, edge set, and conditional probabilities, respectively, from a graph  $G = (V', E')$  in a manner so that the MAP decision problem for the Bayesian network is equivalent to the vertex cover problem on  $G$ .

To begin, we define a function  $B$  that maps a vertex or edge in  $G$  to a vertex in  $BN$ . We also define  $n = |V'|$  and  $m = |E'|$ .

The steps to construct  $BN$  are as follows:

1.  $\forall v \in V', B(v) \rightarrow$  a root node, which we label a V-node, with probability of true/false set to  $(\frac{1}{4} \frac{3}{4})$
2.  $\forall e = (u, v) \in E', B(e) \rightarrow$  an E-node, which has incoming edge from  $B(u)$  and  $B(v)$ , and a probability of an “OR” node, so that this node will be true if either of its parents is true
3. Create 2-input “AND” node between each pair of E-nodes, and then between “AND” nodes, as shown in 5, ending at a final “AND” node  $S$ . We could have made  $S$  an  $m$ -input “AND” node (essentially this is what we are doing), but specifying the distribution as an array would have taken an exponential amount of space, and limits the maximum in-degree of any node to 2. Any node of this type will be true if and only if both of its parents are true. Clearly the number of “AND” nodes is less than  $m$ .
4. Create  $2n$  binary nodes,  $d_1 \dots d_{2n}$  as “probability drain nodes, adding an edge  $s \rightarrow d_i$ . Each of these nodes is true if  $S$  is true, and has a probability of (.5 .5) if  $S$  is false. Notice that if  $S$  is true, then all drain nodes are true. If  $S$  is false, then the probability of each instantiation of  $d_1 \dots d_{2n}$  containing at least one false value is very low. This basically forces  $S$  to be true in a probabilistic manner.

To compute the probability of  $V$  is now (with  $\Phi$  giving us parents of the given node):

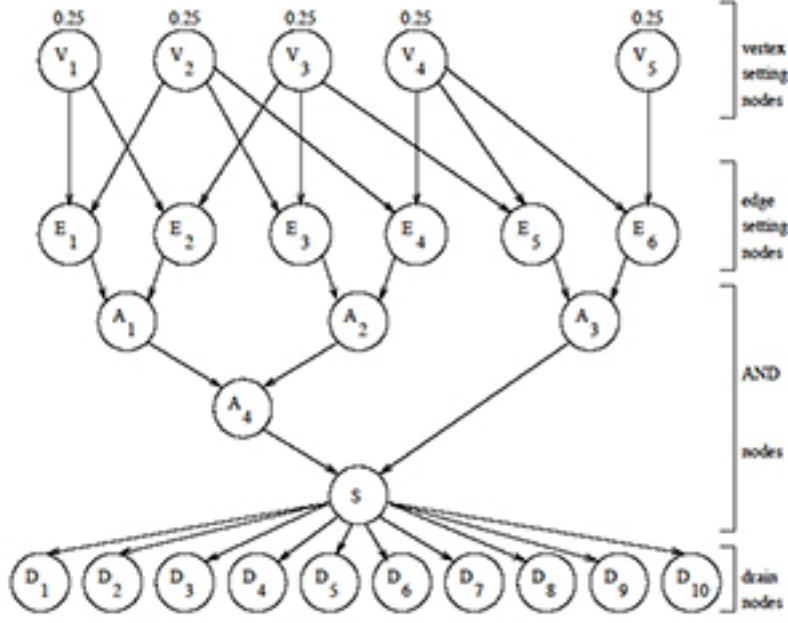


Figure 5: Bayesian network for a vertex cover problem. [11]

$$\begin{aligned}
P(V) &= P(S|\Phi(S)) \\
&\prod_{i=1}^{2n} P(D_i|S) \\
&\prod_{i=1}^m P(AND_i|\Phi(AND_i)) \\
&\prod_{e \in E'} P(B(e)|\Phi(B(e))) \\
&\prod_{v \in V'} P(B(v)) \tag{6}
\end{aligned}$$

The size of the belief network representation of the graph takes time and space polynomial in the size of  $G$ , a necessary condition.

We now show that there is a vertex cover for  $G$  of size  $L \leq K$  if and only if the belief

network we've created has an instantiation  $A$  so that  $P(A) \geq p$  where  $p$  is defined  $p = \frac{3^{n-K}}{4^n}$ .

For  $BN$ , set all non- $V$ -nodes to true. Let  $C$  be a vertex cover for  $G$ . for each  $v \in C$ , set  $B(v) = true$ , and all other  $V$ -nodes to false. Plugging into the equation 6,

$$P(A) = (1)(1)(1)(1) \prod_{v \in V'} P(B(v)) \tag{7}$$

Solving this equation:

$$P(A) = \prod_{v \in V'} P(B(v))$$

$$P(A) = \prod_{v \in C} P(B(v)) \prod_{v \notin C} P(B(v))$$

$$P(A) = \prod_{v \in C} \frac{1}{4} \prod_{v \notin C} \frac{3}{4}$$

$$P(A) = \left(\frac{1}{4}\right)^L \left(\frac{3}{4}\right)^{n-L}$$

$$P(A) = \frac{3^{n-L}}{4^n}$$

and since  $L \leq K$ :

$$P(A) \geq \frac{3^{n-K}}{4^n}$$

$$P(A) \geq p$$

This implies the existence of a vertex cover of at most size  $K$ . For any  $n \geq K \geq 1$ , an instantiation such that  $P(A) \geq p$  must have all nodes set except some V-nodes set to true. If  $S$  is false,  $P(A) < 4^{-n} < p$ , so  $S$  must be true, which causes all other nodes (excluding V-nodes) to be true as well. This implies that the vertices related to the V-nodes that are true form a vertex cover on  $G$ .

If exactly  $L$  V-nodes are true,

$$P(A) = \frac{3^{n-L}}{4^n}$$

$$P(A) \geq \frac{3^{n-K}}{4^n} \text{ only if } L \leq K$$

$$\frac{3^{n-L}}{4^n} \geq \frac{3^{n-K}}{4^n} \text{ only if } L \leq K$$

thus we have a vertex cover of at most  $K$  nodes, which implies that if you can solve MAP Bayesian network decision problem you can solve vertex cover. Clearly then, the decision problem is NP-hard.

## 8 Discussion / Conclusion

In this paper we have introduced Pearl's belief propagation algorithm, and how it is related to the turbo-decoding algorithm. We have, for the general case (and assuming that  $P \neq NP$ ), stated the fact that computing the conditional probabilities, both exactly and approximately, is NP-hard, and have shown a proof that finding the MAPs for Bayesian networks is NP-hard. These results encourage us to look for efficient algorithms for specific types of Bayesian network, such as poly-tree, and not try to solve for the general case. We have also shown that an algorithm that has been shown to be incorrect in certain cases may still present a viable solution to real-world problems, as belief propagation has for turbo codes.

## A Program

1. Program and code available at <http://www.ccs.neu.edu/home/cdb/BayesNet> (please refer to README file at that location)
2. Experiments were restricted to binary (true false) values

## References

- [1] S. A. Barbulescu and S. S. Pietrobon. Turbo codes: A tutorial on a new class of powerful error correcting coding schemes, parts 1 & 2. *Journal of Electri-*

- cal & Electronics Engineering, Australia*, 19(3):129–152, September 1999.
- [2] A. Glavieux C. Berrou and P. Thitimajshima. Near shannon limit error correcting coding and decoding: Turbo codes. *Proceedings IEEE International Communications Conference '93*, 1993.
- [3] G.F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.
- [4] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial Intelligence*, 60:141–153, 1993.
- [5] Erico Guizzo. Closing in on the perfect code. *IEEE Spectrum*, pages 36–42, March 2004.
- [6] Haipeng Guo and William Hsu. A survey of algorithms for real-time bayesian network inference.
- [7] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. pages 467–475, 1999.
- [8] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [9] D. J. MacKay R. J. McEliece and J. F. Cheng. Turbo decoding as an instance of pearl’s belief propagation algorithm. *IEEE Journal of Selected Areas of Communication*, pages 140–152, February 1998.
- [10] E. Rodemich R.J. McEliece and J.F. Cheng. The turbo decision algorithm. *Proceedings 33rd Allerton Conference on Communications, Control and Computing*, 1995.
- [11] S. E. Shimony. Finding maps for belief networks is np-hard. *Artificial Intelligence*, 68:399–410, 1994.
- [12] Yair Weiss and William T. Freeman. Correctness of belief propagation in gaussian graphical models of arbitrary topology. *Neural Computation*, 13(10):2173–2200, 2001.
- [13] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *NIPS*, pages 689–695, 2000.