Penniless Propagation in Join Trees*

Andrés Cano,[1,]† Serafín Moral,[1] Antonio Salmerón[2]
[1] *Department Computer Science and Artificial Intelligence, University of Granada, Spain*
[2] *Department Statistics and Applied Mathematics, University of Almería, Spain*

This paper presents non-random algorithms for approximate computation in Bayesian networks. They are based on the use of probability trees to represent probability potentials, using the Kullback-Leibler cross entropy as a measure of the error of the approximation. Different alternatives are presented and tested in several experiments with difficult propagation problems. The results show how it is possible to find good approximations in short time compared with Hugin algorithm. © 2000 John Wiley & Sons, Inc.

1. INTRODUCTION

Bayesian networks are efficient representations of joint probability distributions, where the network structure encodes the independence relations among the variables. Some different tasks can be performed over Bayesian networks. One of the most common is the computation of posterior marginals given that the value of some variables is known. This task is called *probability propagation*.

Many schemes for the exact computation of such marginals by local computations have been proposed in the last years.[1–5] Local computation consists of calculating the marginals without actually computing the joint distribution, and is described in terms of a message passing scheme over a structure called *join tree*. However, if the model is complicated enough, these schemes may require such a big amount of resources (computing time and memory requirements) that their use becomes difficult.

This fact motivated the development of approximate propagation algorithms. An important group of approximate models is based on Monte Carlo simulation. The idea is to simulate a sample of the variables in the network that is used to estimate the posterior marginals.[6–12]

* This work has been supported by CICYT under projects TIC97–1135–C04–01 and T97–1135–C04–02.

† Author to whom correspondence should be addressed; e-mail: acu@decsai.ugr.es.

Also, deterministic procedures have been proposed for approximate propagation. We can find three main proposals: removing low probability values and considering them to be equal to zero, in order to reduce strong requirements,[13] reducing the complexity of the model by removing weak dependencies[14] and enumerating the configurations of the variables in the model with highest probability to obtain approximations of the marginals.[15,16] A hybrid version of this approach is presented in [17].

In this paper, we present a deterministic approximate algorithm. The algorithm performs a Shenoy-Shafer message passing over a binary join tree,[3] but we introduce the use of probability trees[10] to represent both the messages and the potentials stored in the nodes of the join tree. The use of probability trees will allow to approximate big messages by smaller ones, which makes this algorithm able to run under limited resources or over very large networks.

The reason to use probability trees is that they provide a very general approach to approximate probability potentials. The basic procedure will be to reduce the size of the tree by collapsing several of its branches in only one of them having as value the average of the removed ones. Removing weak dependencies[14] can be seen as a particular case of this operation in which the branches to be collapsed are those coming from tree nodes containing the variables that are not going to be considered in some given dependence relationships. Removing configurations of very low probability[13] can be seen as transforming branches with low value into zero and then applying the basic procedure by reducing several 0 values in only one. This approach is similar to reduce low values to their average, however approximating by 0 should give poorer results than approximating by the average. Finally, by using a propagation scheme we can expect to represent a much greater number of configurations with high probability due to the combinatorial power of local representations with respect to global ones as those used in Ref. 17. Taking into account these considerations, we hope to improve existing approximation algorithms by considering a global approach to obtain good or near to optimal approximations to probability potentials by using probability trees.

The paper is organized as follows: in Section 2, we describe construction of binary join trees and the propagation of probabilities over them using the Shenoy-Shafer scheme; in Section 3, we introduce the new algorithm, called *penniless propagation*; Section 4 is devoted to investigate the use of probability trees to approximate messages, and the results of some experiments carried out over different examples are reported in Section 5; the paper ends suggesting some future works in Section 6.

2. PROPAGATION OVER BINARY JOIN TREES

2.1. Notation

A *Bayesian network* is a directed acyclic graph where each node represents a random variable, and the topology of the graph shows the independence relations among the variables, according to the d-separation criterion.[18] Given

the independences encoded by the graph, the joint distribution is determined giving a probability distribution for each node conditioned on its parents.

Let $X = \{X_1, \ldots, X_n\}$ be the set of variables in the network. Assume each variable X_i takes values on a finite set U_i. For any set U, $|U|$ stands for the number of elements it contains. If I is a set of indices, we will write X_I for the set $\{X_i \mid i \in I\}$. $N = \{1, \ldots, n\}$ will denote the set of indices of all the variables in the network; thus, $X_N = X$. We will denote by U_I the Cartesian product $\Pi_{i \in I} U_i$. Given $x \in U_I$ and $J \subseteq I$, x_J will denote the element of U_J obtained from x dropping the coordinates not in J.

A potential ϕ defined on U_I will be a mapping $\phi : U_I \to \mathbb{R}_0^+$, where $\mathbb{R}_0^+$ is the set of non-negative real numbers. Probabilistic information (including '*a priori*', conditional, and '*a posteriori*' distributions) will always be represented by means of potentials, as in Ref. 2.

By the *size* of a potential, we mean the higher number of values necessary to completely specify it. That is, if ϕ is defined on U_I, its size will be $|U_I|$.

If ϕ is a potential defined on U_I, $s(\phi)$ will denote the set of indices of the variables for which ϕ is defined (i.e., $s(f) = I$).

The *marginal* of a potential ϕ over a set of variables X_J with $J \subseteq I$ is denoted by $\phi^{\downarrow J}$. The conditional distribution of each variable X_i, $i = 1, \ldots, n$, given its parents in the network, $X_{F(i)}$, is denoted by a potential $p_i(x_i \mid x_{F(i)})$ where p_i is defined over $U_{\{i\} \cup F(i)}$.

Then, the joint probability distribution for the n-dimensional random variable X_N can be expressed as

$$p(x) = \prod_{i \in N} p_i(x_i \mid x_{F(i)}) \quad \forall x \in U_N \tag{1}$$

An *observation* is the knowledge about the exact value $X_i = e_i$ of a variable. The set of observations will be denoted by e, and called the *evidence set*. E will be the set of indices of the observed variables.

Every observation, $X_i = e_i$, is represented by means of a potential which is a Dirac function defined on U_i as $\delta_i(x_i; e_i) = 1$ if $e_i = x_i$, $x_i \in U_i$, and $\delta_i(x_i; e_i) = 0$ if $e_i \neq x_i$.

The goal of probability propagation is to calculate the '*a posteriori*' probability function $p(x'_k \mid e)$, for every $x'_k \in U_k$, where $k \in \{1, \ldots, n\} - E$.

It is well known that

$$p(x'_k \mid e) \propto \sum_{x_k = x'_k} \left(\prod_i p_i(x_i \mid x_{F(i)}) \prod_{e_i \in e} \delta_i(x_i; e_i) \right) \tag{2}$$

If we call $H = \{p_i(x_i \mid x_{F(i)}) \mid i = 1, \ldots, n\} \cup \{\delta_i(x_i; e_i) \mid e_i \in e\}$, and

$$\phi_k^m = \left(\prod_{\phi \in H} \phi \right)^{\downarrow k} \tag{3}$$

then according to equation (2), the desired conditional probability can be computed by obtaining ϕ_k^m according to equation (3) and normalizing it afterwards in such a way that the sum of its values is equal to 1.

The functions in H can be simplified taking into account the observations in e. If each $\phi \in H$ such that ϕ is not an observation, it is transformed into a function ϕ_e defined on $s(\phi) - E$ and given by:

$$\phi_e(x) = \phi(y), \text{ where } y_{s(\phi)-E} = x, \text{ and } y_i = e_i, \forall i \in E \tag{4}$$

That is, for every function F we remove variables in E by fixing them to the observed values. If H_e is the set obtained from H by changing every non-delta potential ϕ to ϕ_e, then we still continue having an equation similar to equation (3):

$$\phi_k^m = \left(\prod_{\phi \in H_e} \phi \right)^{\downarrow k} \tag{5}$$

In this paper, we will use these reduced potentials. The advantage is that they are more simple than initial potentials in H. The disadvantage is that all posterior computations will depend on the observations and therefore they cannot be reused if the set of observations changes.

The computation of ϕ_k^m is usually organized in a join tree. A *join tree* is a tree where each node V is a subset of X_N, and such that if a variable is in two distinct nodes, V_1 and V_2, then it is also in every node in the path between V_1 and V_2. This property is known as *junction property*.[19] A join tree is called *binary* if every node has no more than three neighbors.

Every potential, $\phi \in H_e$ is assigned to a node V_j such that $s(\phi) \subseteq V_j$. In this way, attached to every node V_i there will be a potential ϕ_i defined over the set of variables V_i and that is equal to the product of all the potentials assigned to it.

By means of a propagation algorithm in a join tree ϕ_k^m can be calculated. After it, ϕ_k^m can be obtained from any node V_j containing variable X_k.

2.2. Constructing a Binary Join Tree

The process of constructing a join tree from a Bayesian network can be divided into two stages: first, determine which variable will form each node in the tree, and second, arrange the nodes in such a way that the junction property is verified.

The first task is usually performed by triangulating the undirected graph associated to the set of potentials H_e (two variables are joined if and only if there is a potential $\phi \in H_e$ such that both variables are in $s(\phi)$), and then obtaining clusters of variable which are pairwise connected of the triangulated graph.[19] The groups are not necessarily cliques, i.e., maximal completely connected clusters, but all the cliques should be considered. At the end, each cluster corresponds to a node in the join tree. The triangulation of the graph can be carried out by removing the nodes in sequence, and each time a node is deleted, a cluster is formed by all the nodes that are neighbors of the one being removed.

Different deletion sequences result in different sets of clusters, and we will prefer those sets whose clusters are of smaller size, or, more precisely, those whose corresponding potentials are of smaller size. Finding the optimal deletion sequence is not a trivial task, and several heuristics have been proposed.[20,21] One of the most popular heuristics is the *minimum size criterium*, which consists of selecting, at each step, the variable producing the smallest cluster. Here we adopt a variation over it: we first remove variables which are not observed and whose descendants are not observations either, starting from leaves upward. This will have implications in the resulting propagation algorithms: upward messages from the clusters produced when deleting these variables will be the result of marginalizing a conditional probability $p_i(x_i|x_{F(i)})$ by adding on x_i. The result will be a potential which is identically equal to 1, and then easily representable by probability trees with only one value. Doing this, we can obtain some clusters larger than in the usual criterium, but some of the messages will be extremely simple. It is necessary to arrange the clusters in the tree in an order opposite to that produced by the elimination sequence, that is, the last cluster produced will be the first inserted to the tree and so on.

A detailed algorithm to compute a join tree from a Bayesian network can be as follows:

ALGORITHM JoinTree($\mathscr{B}$)

INPUT: An undirected graph $\mathscr{B}$.
OUTPUT: $\mathscr{J}$, a join tree obtained from network $\mathscr{B}$.

1. Select a deletion sequence according to the modified minimum size criterium.
2. Let $\mathscr{L} = \{C_1, \ldots, C_m\}$ be the list of clusters corresponding to the elimination sequence.
3. Extract cluster C_m from $\mathscr{L}$.
4. Let $\mathscr{J}$ be a join tree whose only node is C_m.
5. FOR $i := m - 1$ TO 1
   - Extract cluster C_i from $\mathscr{L}$.
   - Let V be the set of variables in C_i that are also contained in any node in the join tree.
   - Connect cluster C_i to a node in $\mathscr{J}$ that contains V. If no node in $\mathscr{J}$ contains V, leave C_i unconnected.
6. RETURN $\mathscr{J}$.

Example 1. Consider the undirected graph in Figure 1, and assume all the variables are binary. A possible deletion sequence according to the modified minimum size criterium is X_5, X_4, X_3, X_2, X_1. The clusters resulting from this elimination sequence are $\{X_5, X_4, X_3\}$, $\{X_4, X_3, X_2\}$, $\{X_3, X_2, X_1\}$, $\{X_2, X_1\}$. To construct the join tree, we select the last cluster obtained: $\{X_2, X_1\}$, resulting in a join tree with a single node containing variables X_2 *and* X_1 [see Fig. 2(a)].

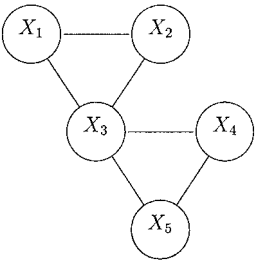

Figure 1. An undirected graph.

Now we select the next cluster, $\{X_3, X_2, X_1\}$. The intersection of its variables with the variables already contained in the join tree is $\{X_2, X_1\}$. Thus, we must connect this cluster to a node containing variables X_2 and X_1 [Fig. 2(b)]. In the next step, we choose $\{X_4, X_3, X_2\}$. In this case, the intersection is $\{X_3, X_2\}$, which implies that the cluster has to be connected as in Figure 2(c). Finally, we insert cluster $\{X_5, X_4, X_3\}$, obtaining the join tree in Figure 2(d).

Observe that the join tree provided by the algorithm above is not always binary. When applying the Shafer and Shenoy[4] propagation scheme, the use of binary trees is very suitable.[3] The reason is that they store intermediate results so that they do not have to be recalculated when used in different computations. Furthermore, it is straightforward to convert a joint tree into a binary one applying the following algorithm.

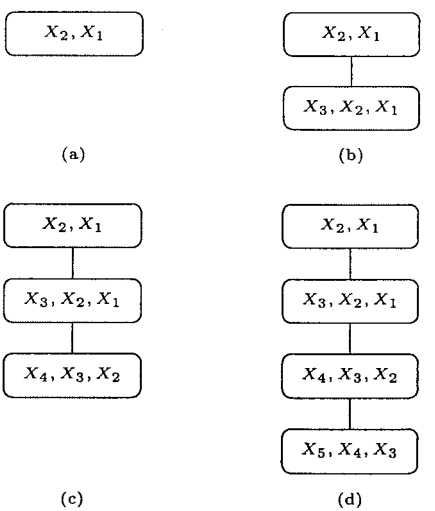

Figure 2. Construction of a join tree.

ALGORITHM Binary($\mathcal{J}$)

INPUT: A join tree $\mathcal{J}$.
OUTPUT: The input join tree $\mathcal{J}$ in binary format.

1. WHILE there is a node V in $\mathcal{J}$ with more than three neighbors,
   - Let V_1, V_2 be two neighbors of V.
   - Create a new node $V_3 = (V_1 \cup V_2) \cap V$.
   - Connect V_3 to V.
   - Disconnect V_1 and V_2 from V and connect them to V_3.
2. RETURN $\mathcal{J}$.

Example 2. Figure 3 illustrates this algorithm. Nodes $\{X_1, X_3\}$ and $\{X_1, X_4\}$ are disconnected from $\{X_1, X_2\}$ and connected to a new node $\{X_1\}$. This new node is then connected to $\{X_1, X_2\}$.

Now that we have a binary join tree, the next step is to initialize the probabilistic information it must contain. To this end, each probability potential present in the original Bayesian network must be included in at least one cluster. The other clusters will contain a unit potential, i.e., a potential that is constantly equal to 1.

2.3. Shenoy-Shafer Propagation over Join Trees

In this scheme, two mailboxes are placed on each edge of the join tree. Given an edge connecting nodes V_i and V_j, one mailbox is for messages V_i-outgoing and V_j-incoming, and the other mailbox is for the reverse. The messages allocated in both mailboxes will be probability potentials defined on $V_i \cap V_j$. Initially, all mailboxes are *empty*, and once a message has been placed on one of them, it is said to be *full*.

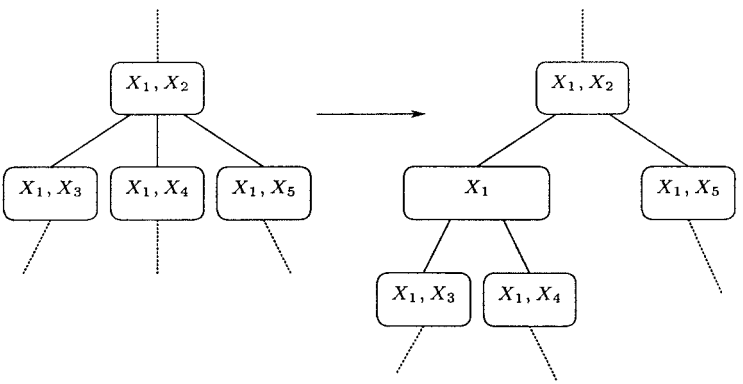

Figure 3. Making a join tree be binary.

A node V_i in a join tree is allowed to send a message to its neighbor node V_j if and only if all V_i-incoming mailboxes are full except the one from V_j to V_i. Thus, initially only nodes corresponding to leaves can send messages. The message V_i-outgoing and V_j-incoming is computed as

$$\phi_{V_i \to V_j} = \left\{ \phi_i \cdot \left(\prod_{V_k \in ne(V_i) - V_j} \phi_{V_k \to V_i} \right) \right\}^{\downarrow V_i \cap V_j} \tag{6}$$

where ϕ_i is the initial probability potential on V_i, $\phi_{V_k \to V_i}$ are the messages in the mailboxes V_k-outgoing and V_i-incoming and $ne(V_i)$ are the neighbor nodes of V_i. Note that one message contains the information coming from one side of the tree and is sent to the other side of the tree. It can be shown[4] that there is always at least one node allowed to send a message until all mailboxes are full, and when the message passing ends, for every node V_i in the joint tree it holds that

$$\phi_{V_i}^m = \phi_i \cdot \left(\prod_{V_k \in ne(V_i)} \phi_{V_k \to V_i} \right) \tag{7}$$

where $\phi_{V_i}^m = (\prod_{\phi \in H_e} \phi)^{\downarrow V_i}$ is proportional to the conditional distribution of the variables in V_i given observation e. The desired conditional probability for variable X_k and ϕ_k^m can be calculated by marginalizing $\phi_{V_i}^m$ over this variable and normalizing the result.

The propagation can be organized in two stages. In the first one, messages are sent from leaves to a previously selected root node, and in the second stage, messages are sent from the root to the leaves. The following algorithm implements this propagation scheme:

ALGORITHM Shenoy-Shafer ($\mathcal{J}$)

INPUT: A join tree $\mathcal{J}$.
OUTPUT: Join tree $\mathcal{J}$ after propagation.

1. Select a root node R.
2. FOR each $V \in Ne(R)$,
   - **NavigateUp**(R, V)
3. FOR each $V \in ne(R)$,
   - Compute message

$$\phi_{R \to V} = \left\{ \phi_R \cdot \left(\prod_{V_k \in ne(R) - V} \phi_{V_k \to R} \right) \right\}^{\downarrow R \cap V}.$$

   - **NavigateDown**(R, V)
4. RETURN $\mathcal{J}$.

where **NavigateUp** is a procedure that sends messages from leaves to root and **NavigateDown** sends messages from root to leaves. Both procedures are detailed below.

NavigateUp(V_1, V_2)

1. FOR each $V \in ne(V_2) \to V_1$,
   - **NavigateUp**(V_2, V)
2. Compute message

$$\phi_{V_2 \to V_1} = \left\{ \phi_2 \cdot \left(\prod_{V_k \in ne(V_2) - V_1} \phi_{V_k \to V_2} \right) \right\}^{\downarrow V_1 \cap V_2}.$$

NavigateDown(V_1, V_2)

1. FOR each $V \in ne(V_2) - V_1$,
   - Compute message

$$\phi_{V_2 \to V} = \left\{ \phi_2 \cdot \left(\prod_{V_k \in ne(V_2) - V_1} \phi_{V_k \to V_2} \right) \right\}^{\downarrow V \cap V_2}.$$

   - **NavigateDown**(V_2, V)

The sequence of message passing is illustrated in Figure 4.

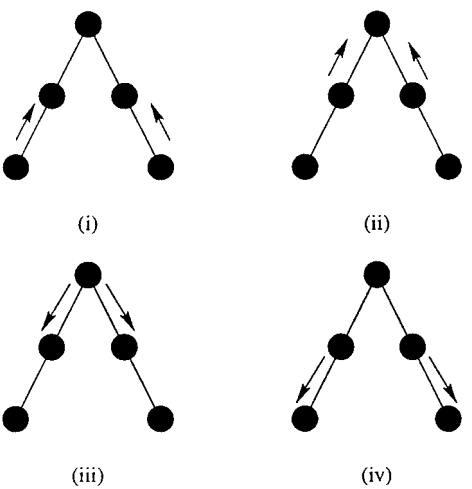

Figure 4. Message passing in the Shenoy-Shafer propagation.

3. PENNILESS PROPAGATION

The Shenoy-Shafer algorithm can be used to obtain the posterior marginal distribution for all the variables in the network. However, it may be infeasible to obtain those marginals in a reasonable time. For instance, assume we have to compute a message defined over 14 variables, each of them taking three possible values, and that we use probability tables to handle the probabilistic information about the variables; in this case, to specify the corresponding probability potential implies to compute and store 3^{14} values, which using 8 bytes real numbers requires more than 36 MBytes of memory, and this is just for one among the hundreds of messages that may appear when propagating over a large network.

Besides, even if the network is not that large, in some cases we have to perform the propagation over a computer with limited resources (small amount of RAM, slow CPU). In this situation, we will be forced to use approximate methods which, in exchange of losing the precision in the computations, provide results in more reasonable time.

Our proposal in this work consists of a propagation algorithm, based on Shenoy-Shafer's, but able to provide results under limited resources. To achieve this, we will assume that messages are represented by means of probability trees.[2] This tool provides an approximate representation of potentials within a given maximum number of values.[23–25,10]

The consequence is that the messages that are sent during the propagation will be approximated if their size is greater than a given threshold. This threshold will depend on the available resources. In Section 4 we discuss how this approximation is carried out and how the operations over probability trees are performed.

Another difference with respect to Shenoy-Shafer's algorithm is the number of stages of the propagation. In the exact algorithm, there are two stages: in the first one, messages are sent from leaves to the root and in the second one, in the opposite direction.

Here we add the possibility of performing more than two stages. The goal is to improve the approximate messages at each stage, by taking into account messages coming from other parts of the join tree. More precisely, when a message is sent throw an edge, it is approximated conditional on the message contained in the same edge but in the opposite direction. The idea of conditional approximation of potentials was proposed in 23, 26. The reason is the following: when we are going to approximate a message, for example $\phi_{V_i \to V_j}$, the first idea is to try to find a potential $\phi^0_{V_i \to V_j}$ such that the distance $D(\phi_{V_i \to V_j}, \phi^0_{V_i \to V_j})$ is minimum. The distance commonly used is the Kullback-Leibler cross entropy[27] between the normalized potentials. But our final objective is not to compute messages but to compute marginal probabilities. After propagation, the marginal probability for variables in $V_i \cap V_j$ will be proportional to $\phi_{V_i \to V_j} \cdot \phi_{V_j \to V_i}$. So we should carry out the approximation of a

[2] Initially if no message has been sent this message is the potential equal to 1 for all the configurations.

message trying to minimize the value of conditional information, given by:

$$D\left(\phi_{V_i \to V_j}, \phi^0_{V_i \to V_j} \mid \phi_{V_j \to V_i}\right) = D\left(\phi_{V_i \to V_j} \cdot \phi_{V_j \to V_i}, \phi^0_{V_i \to V_j} \cdot \phi_{V_j \to V_i}\right) \quad (8)$$

This conditioning will equalize the distance between $\phi_{V_i \to V_j}$ and $\phi^0_{V_i \to V_j}$ to give more importance to the differences between the message and its approximation in those cases in which $\phi_{V_j \to V_i}$ is high. In an extreme situation, imagine that $\phi_{V_j \to V_i}$ is 0 for all the elements of $A \subseteq U_{V_j \cap V_i}$, then the differences between $\phi_{V_i \to V_j}$ and its approximation in set A will not be relevant: we can change the values of $\phi_{V_i \to V_j}$ on A having always the same distance. This has perfect sense if we take into account that the result of the conditional probability will be 0 in this set whatever the value of $\phi_{V_i \to V_j}$ is.

The problem is that when we send a message $\phi_{V_i \to V_j}$, usually we do not have the opposite message, $\phi_{V_j \to V_i}$, but an approximation[2] of it, $\phi^0_{V_j \to V_i}$, and the distance has to be conditioned to this approximation. When we first computed $\phi^0_{V_j \to V_i}$, $\phi^0_{V_i \to V_j}$ was not available. Once we have it, we can use it to compute a better approximation of $\phi_{V_j \to V_i}$: the message $\phi^1_{V_j \to V_i}$ such that $D(\phi_{V_j \to V_i}, \phi^1_{V_j \to V_i} \mid \phi^0_{V_i \to V_j})$ is minimum. But now that we have a better approximation of $\phi_{V_j \to V_i}$, we could use it as the conditioning part in formula (8) to obtain a better approximation of $\phi_{V_i \to V_j}$. This process can continue until no further change is obtained in the messages. No theoretical result is available at this moment about the convergence of this procedure, but in Refs. 23, 26 there are some experiments in which the convergence is very fast (a few iterations).

If in a given moment some message, say $\phi_{V_i \to V_j}$, is exact, then there is no need to try to improve it by iterating as described above. We have organized our algorithm so that it carries out several iterations, but only two of them (first and last) are necessary in all the messages, the rest will be carried out only through approximate messages. With this aim, we distinguish three different types of stages: the first stage, the intermediate ones and the last stage.

The goal of the first stage is to collect information from the entire join tree in the root node, in order to distribute it to the rest of the graph in a posterior stage. Messages are sent from leaves to the root, approximating those whose size is too big. We keep a flag in every message indicating whether its computation was exact or approximate. This information will be used in the intermediate stages.

In the intermediate stages, we try to improve the messages stored in each edge, performing several runs over the join tree and updating the approximate messages according to the information coming from other parts of the tree. To achieve this, messages are sent in both directions: first, from the root towards the leaves and second, from the leaves upwards. When a message is going to be sent through an edge, we check the flag of the message in the opposite direction; if that message is labeled as exact, we do not continue updating the messages in the subtree determined by that edge. The reason to do this is that sending messages through that edge will not help to better approximate the messages in the opposite directions, since those messages are already exact, so we do not need to update them.

In the last stage the propagation is completed sending messages from the root to the leaves. In this case, messages are sent downward even if the message

stored in the opposite direction is exact. This is done to assure that at the end of the propagation all the clusters in the join tree have received the corresponding messages.

In the next we describe the detailed algorithm, called PENNILESS PROPAGATION, emphasizing the fact that it performs a propagation even in absence of a big amount of resources.

ALGORITHM PENNILESS PROPAGATION ($\mathcal{J}$, *stages*)

INPUT: A join tree $\mathcal{J}$ and the number of propagation stages.
OUTPUT: Join tree $\mathcal{J}$ after propagation.

1. Select a root node R.
2. **NavigateUp**(R)
3. $stages := stages - 1$
4. WHILE $stages > 2$
   NavigateDownUp(R)
   $stages := stages - 2$
5. IF ($stages == 1$)
   NavigateDown(R)
   ELSE
   NavigateDownUpForcingDown(R)

The first stage is carried out by calling procedure **NavigateUp**. This procedure requests a message from each one of its neighbors, which recursively do the same to all its neighbors downwards until the leaves are reached. In this moment, messages are sent upwards until the root is reached. This task is implemented by the following two procedures.

NavigateUp(R)

1. FOR each $V \in ne(R)$,
   NavigateUp(R, V)

NavigateUp(S, T)

1. FOR each $V \in ne(T) - \{S\}$,
   - **NavigateUp**(T, V)
2. **SendApprMessage**(T, S)

Once the root node has received messages from all its neighbors, the intermediate stages begin. This is carried out by the next two procedures. Observe that exact messages are not updated.

NavigateDownUp(R)

1. FOR each $V \in ne(R)$
   IF $\phi_{V \to R}$ is not exact
   IF $\phi_{R \to V}$ is not exact
   SendApprMessage(R, V)
   NavigateDownUp(R, V)

NavigateDownUp(S, T)

1. FOR each $V \in ne(T) - \{S\}$
   IF $\phi_{V \to T}$ is not exact
   IF $\phi_{T \to V}$ is not exact
   SendApprMessage(T, V)
   NavigateDownUp(T, V)
2. IF $\phi_{T \to S}$ is not exact
   SendApprMessage(T, S)

Finally, the last step is carried out by calling the procedures described next. After this, the posterior marginal for any variable in the network can be computed by selecting any node containing that variable and marginalizing its corresponding potential. Observe that after the intermediate stages, if the total number of stages is odd, we still have to perform two traversals, one downwards and one upwards. The difference with respect to the intermediate steps is that in this case, messages are sent downwards even if they are marked as exact, in order to assure that the posterior marginals can be obtained in any node.

NavigateDownUpForcingDown(R)

1. FOR each $V \in ne(R)$
   IF $\phi_{V \to R}$ is not exact
   IF $\phi_{R \to V}$ is not exact
   SendApprMessage(R, V)
   NavigateDownUpForcingDown(R, V)
   ELSE
   IF $\phi_{R \to V}$ is not exact
   SendApprMessage(R, V)
   NavigateDown(R, V)

NavigateDownUpForcingDown(S, T)

1. FOR each $V \in ne(T) - \{S\}$
   IF $\phi_{V \to T}$ is not exact
   IF $\phi_{T \to V}$ is not exact
   SendApprMessage(T, V)
   NavigateDownUpForcingDown(T, V)
   ELSE
   IF $\phi_{T \to V}$ is not exact
   SendApprMessage(T, V)
   NavigateDown(T, V)
2. **SendApprMessage**(T, S)

However, if the total number of stages is even, in the final stage we just have to send messages downwards, which is implemented by the next procedures.

NavigateDown(R)

1. FOR each $V \in ne(R)$
   SendApprMessage(R, V)
   NavigateDown(R, V)

NavigateDown(S, T)

1. FOR each $V \in ne(T) - \{S\}$
   SendApprMessage(T, V)
   NavigateDown(T, V)

Now we show the details of procedure **SendApprMessage**, used in the algorithms above. A message $\phi_{S \to T}$ is computed by combining all the incoming messages of S except that one coming from T with the potential in S, and then, the resulting potential is approximated, if necessary, according to the message coming from T to S, $\phi_{T \to S}$.

SendApprMessage(S, T)

1. Compute

$$\phi = \left(\prod_{V \in ne(S) - \{T\}} \phi_{V \to S} \right)^{\downarrow S}$$

2. IF at least one of the messages $\phi_{V \to S}$ is not exact, mark ϕ as approximate.
3. Compute $\phi_{S \to T} = \phi \cdot \phi_S$.
4. IF the size of $\phi_{S \to T}$ is too big
   (a) Approximate $\phi_{S \to T}$ conditional on $\phi_{T \to S}$.
   (b) Mark $\phi_{S \to T}$ as approximate.

4. PROBABILITY TREES

A *probability tree*[22,25,10] is a directed labeled tree, where each internal node represents a variable and each leaf node represents a probability value. Each internal node will have as many outgoing arcs as possible values the variable it represents has. Each leaf of the tree contains a real number. We define the *size* of the tree as the number of leaves it has.

A probability tree $\mathcal{T}$ on variables X_I represents potential $\phi : U_I \to \mathbb{R}$ if for each $x_I \in U_I$ the value $\phi(x_I)$ is the number stored in the leaf node that is obtained starting in the root node and selecting for each inner node labeled with X_i the child corresponding to coordinate x_i. The potential represented by tree $\mathcal{T}$ will be denoted by $\phi_{\mathcal{T}}(x_I)$.

Probability trees are appropriate tools for representing regularities in potentials.

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
1	1	1	0.2
1	1	2	0.5
1	2	1	0.7
1	2	2	0.7
2	1	1	0.8
2	1	2	0.5
2	2	1	0.3
2	2	2	0.3

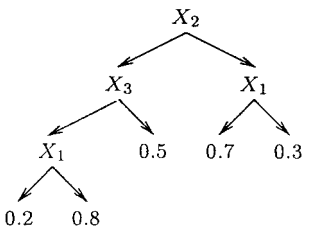

Figure 5. A potential and a probability tree representing it.

Example 3. Figure 5 displays a potential ϕ and a probability tree representing it. The tree contains the same information as the table, but using five values instead of eight.

Each node of the tree and in particular its leaves is characterized by a configuration of values $X_J = x_J$, where $J \subseteq I$. The variables in X_J are the variables in the path from the root to the node and the values of the variables correspond to the branches we have to follow to arrive at the node.

In the next section we briefly describe the construction of probability trees from a specified potential and the basic operations over them. For a more detailed exposition, see Ref. 10. We will also describe how the approximations used in the PENNILESS PROPAGATION algorithm can be carried out.

4.1. Constructing a Probability Tree

Constructing a tree $\mathcal{T}$ representing a potential ϕ for a set of variables X_I without any other additional restriction is rather straightforward: the tree will contain one branch for every configuration of X_I, and one leaf for every value $\phi(x_I)$ with $x_I \in U_I$. However, this procedure can lead to unnecessarily big trees. For example, consider the trees in Figure 6. Assume tree (a) is the result of the procedure above. Changing the positions of variables X_1 and X_2 we obtain tree (b). Now, we can realize that the value of X_1 is irrelevant given the value of X_2. Thus, we can construct tree (c) which represents the same potential as (a) and (b) but with lower size.

An efficient method for constructing a probability tree from a given potential is described in Ref. 10. It is built in an incremental way, by selecting the most informative variable as label for each interior node. The information is measured as the different in Kullback–Leibler distance of the partial tree before and after branching to the complete potential.

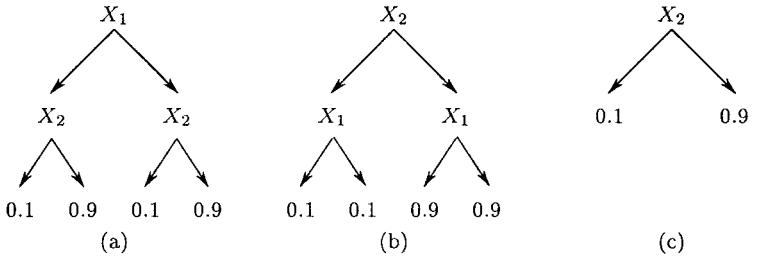

Figure 6. Three equivalent trees.

4.2. Operations Over Probability Trees

Propagation algorithms require two basic operations over potentials: *combination* (product) and *marginalization* (projection). In this section we briefly describe algorithms to perform these operations.[25,10]

A third operation, *restriction*, is also necessary to specify the algorithms in this section: given a tree $\mathcal{T}$, a set of variables X_J, and $x_J \in U_J$, $\mathcal{T}^{R(X_J=x_J)}$ denotes the restriction of $\mathcal{T}$ to the values x_J of the variables in X_J, that is, the tree obtained by substituting in $\mathcal{T}$ every node corresponding to variables X_k, $k \in J$ by the subtrees $\mathcal{T}_k$ children of X_k corresponding to $X_k = x_k$. This operation is illustrated in Figure 7.

We proceed to describe the other two operations, *combination* and *marginalization.*

Given two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ representing potentials ϕ_1 and ϕ_2 respectively, the following algorithm computes a tree representing potentials $\phi = \phi_1 \cdot \phi_2$ (combination).

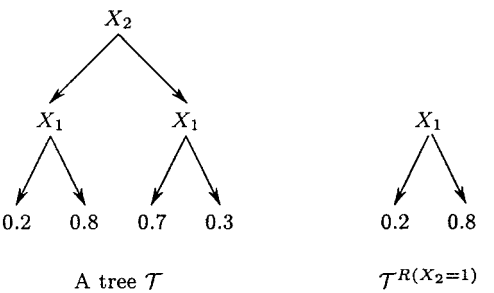

Figure 7. The restriction operation.

COMBINE($\mathcal{T}_1, \mathcal{T}_2$)

1. Create a tree node $\mathcal{T}_r$ initially with no label.
2. Let L_1 and L_2 be the labels of the root nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively.
3. IF L_1 and L_2 are numbers, THEN make $L_1 \cdot L_2$ be the label of $\mathcal{T}_r$.
4. IF L_1 is a number but L_2 is a variable, THEN
   (a) Make L_2 be the label of $\mathcal{T}_r$.
   (b) FOR every tree $\mathcal{T}$ child of the root node of $\mathcal{T}_2$,
   make $\mathcal{T}_h :=$ **COMBINE**($\mathcal{T}_1, \mathcal{T}$) be a child of $\mathcal{T}_r$.
5. If L_1 is a variable, assume that X_k is that variable.
   (a) Make X_k be the label of $\mathcal{T}_r$.
   (b) FOR each $x_k \in U_k$,
   • Make $\mathcal{T}_h :=$ **COMBINE**($\mathcal{T}_1^{R(X_k = x_k)}, \mathcal{T}_2^{R(X_k = x_k)}$) be a child of $\mathcal{T}_r$.
6. RETURN $\mathcal{T}_r$.

We will denote the combination of trees by symbol $\otimes$. With this notation, the algorithm above returns a tree $\mathcal{T}_r = \mathcal{T}_1 \otimes \mathcal{T}_2$.

The combination process is illustrated in Figure 8. Given a tree $\mathcal{T}$ representing a potential ϕ defined over a set of variables X_I, the following algorithm computes a tree representing potential $\phi^{\downarrow (I - \{i\})}$, with $i \in I$. That is, it removes variable X_i from $\mathcal{T}$.

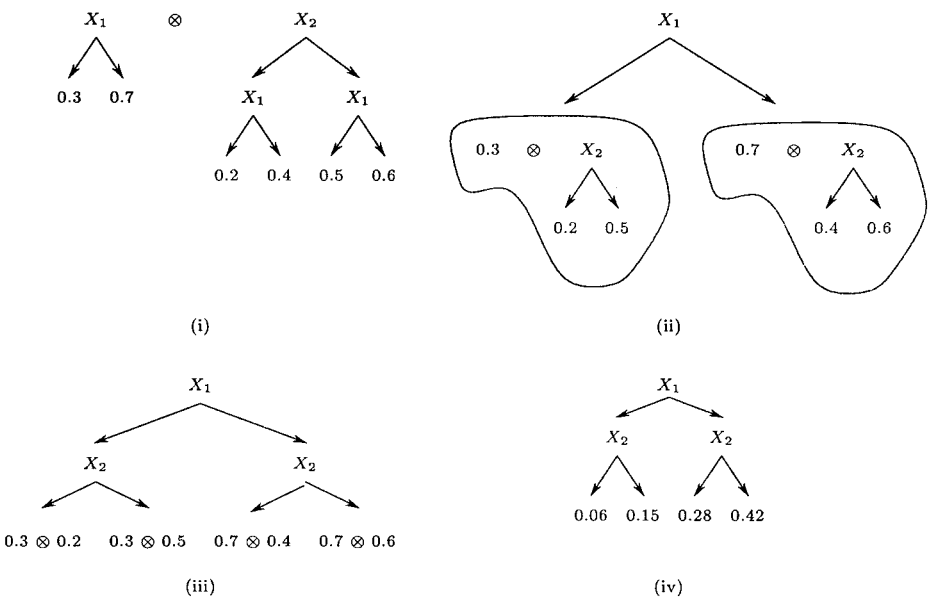

Figure 8. Combination of two trees.

MARGINALIZE($\mathcal{T}$, X_i)

1. Let L be the label of the root node of $\mathcal{T}$.
2. If L is a number, create a node $\mathcal{T}_r$ with label $L \cdot |U_i|$.
3. Otherwise, let X_k be the variable corresponding to label L.
   (a) IF $X_k = X_i$, THEN
      i. Let $\mathcal{T}_1, \ldots, \mathcal{T}_s$ be the children of the root node of $\mathcal{T}$.
      ii. $\mathcal{T}_r := \mathcal{T}_1$.
      iii. For $i := 2$ to s, $\mathcal{T}_r := \textbf{ADD}(\mathcal{T}_r, \mathcal{T}_i)$.
   (b) ELSE
      i. Create a node $\mathcal{T}_r$ with label X_k.
      ii. FOR each $x_k \in U_k$
         A. Make $\mathcal{T}_h := \textbf{MARGINALIZE}(\mathcal{T}^{R(X_k = x_k)}, X_i)$ be the next child of $\mathcal{T}_r$.
4. RETURN $\mathcal{T}_r$.

This algorithm uses procedure **ADD**($\mathcal{T}_1$,$\mathcal{T}_2$) which computes the addition of $\mathcal{T}_1$ and $\mathcal{T}_2$. The procedure is as follows:

ADD($\mathcal{T}_1$,$\mathcal{T}_2$)

1. Create a tree node $\mathcal{T}_r$ initially with no label.
2. Let L_1 and L_2 be the labels of the root nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively.
3. IF L_1 and L_2 are numbers, THEN make $L_1 + L_2$ be the label of $\mathcal{T}_r$.
4. IF L_1 is a number but L_2 is a variable, THEN
   (a) Make L_2 be the label of $\mathcal{T}_r$.
   (b) For every child $\mathcal{T}$ of the root node of $\mathcal{T}_2$, make $\mathcal{T}_h := \textbf{ADD}(\mathcal{T}_1, \mathcal{T})$ be a child of $\mathcal{T}_r$.
5. IF L_1 is a variable, assume that X_k is that variable.
   (a) Make X_k be the label of $\mathcal{T}_r$.
   (b) FOR each $x_k \in U_k$,
      • Make $\mathcal{T}_h := \textbf{ADD}(\mathcal{T}_1^{R(X_k = x_k)}, \mathcal{T}_2^{R(X_k = x_k)})$ be a child of $\mathcal{T}_r$.
6. RETURN $\mathcal{T}_r$.

The addition of two trees is illustrated in Figure 9, where symbol $\oplus$ represents the addition operation.

4.3. Approximate Probability Trees

Now consider that we use probability trees to implement the PENNILESS PROPAGATION algorithm. According to procedure **SendApprMessage**, when a message is sent from a node V_i to another node V_j, all the messages that come to V_i except $\phi_{V_j \to V_i}$ are combined, collecting all the information from that part of the join tree that is relevant to the variables in V_j (see Fig. 10). After the combination of the messages and the marginalization on variables of $V_i \cap V_j$, the tree representing potential $\phi_{V_i \to V_j}$ may be too big (i.e., it may require a big amount of numbers to be represented).

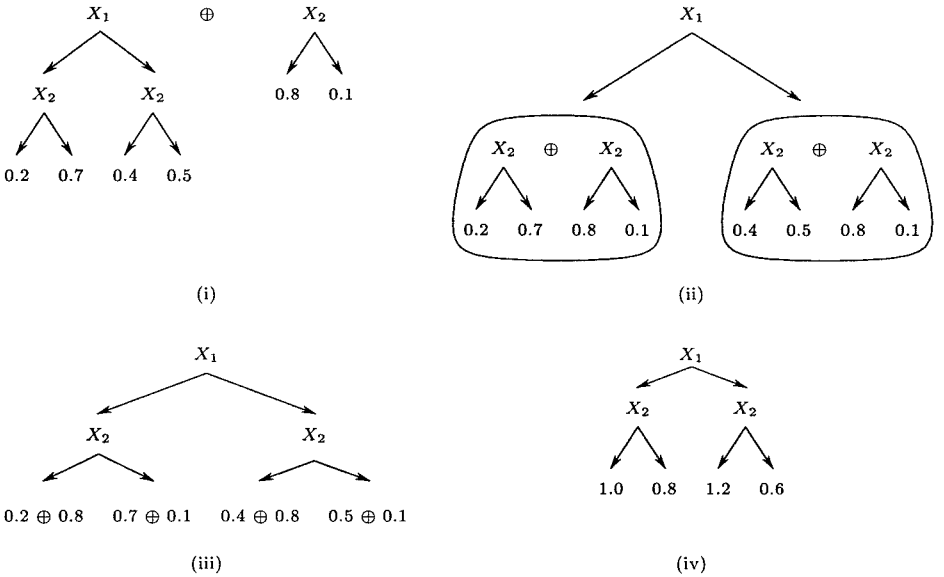

Figure 9. Addition of two trees.

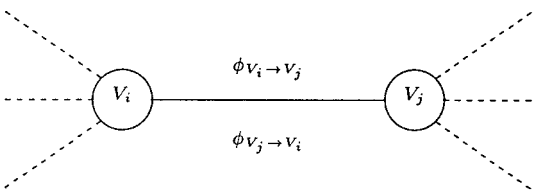

Figure 10. Messages throw an edge.

In this situation, we will have to reduce the size of the probability tree, removing some nodes from it and obtaining thus, an approximate probability tree, in the sense that it represents a potential that is not exactly $\phi_{V_i \to V_j}$, but an approximation.

In general, our problem will be the following: to approximate a potential ϕ represented by a tree $\mathcal{T}$, by another potential ϕ' represented by another tree $\mathcal{T}'$ of smaller size, conditional on another potential ψ. All the potentials will be assumed to be defined on frame U_I. For a potential ϕ, let us consider the following notation:

- $\text{sum}(\phi, A) = \sum_{x \in A} \phi(x)$, where $A \subseteq U_I$.
- $\text{sum}(\phi) = \text{sum}(\phi, U_I) = \sum_{x \in U_I} \phi(x)$
- If $\text{sum}(\phi) \neq 0$, then $N(\phi) = \frac{\phi}{sum(\phi)}$

We will measure the distance between two potentials ϕ and ϕ' conditional on ψ by the Kullback-Leibler cross entropy between the normalized potentials:

$$D(\phi, \phi' \mid \psi) = \sum_{x \in U_I} N(\phi(x) \cdot \psi(x)) \log\left(\frac{\phi(x)\text{sum}(\phi' \cdot \psi)}{\phi'(x)\text{sum}(\phi \cdot \psi)} \right) \tag{9}$$

Since there is no difference between the distances $D(\phi, \phi' \mid \psi)$ and $D(\phi, \phi'' \mid \psi)$ if $N(\phi') = N(\phi'')$, i.e., the distance is independent of the normalization factor, then ϕ' will be determined up to a normalization value. In this paper we shall consider that the approximations ϕ' of ϕ are such that $\text{sum}(\phi') = \text{sum}(\phi)$, i.e., the normalization value of the approximation will be the same as the one of the exact potential.

As it has been reported in Refs. 22, 25, the difficulty of the approximation lies in finding the structure of the tree, i.e. the same tree without numbers on the leaves. Given a structure $\mathcal{S}$ we can build an approximate tree denoted by $\mathcal{T}_{\mathcal{S}}$ of ϕ by assigning to each leave characterized by configuration $X_J = x_J$ the average of potential ϕ in the points $y \in U_I$ such that they coincide with x_J in the values of variables X_J: $y_J = x_J$. In this way, we also have the same normalization values for the original and the approximate trees.

As finding an optimal tree structure is a NP-hard problem, most of the procedures modify a given structure in an incremental way following some heuristics. If we have a structure $\mathcal{S}'$, we have two basic modifications:

- *Branching*–If we have a leave node characterized by configuration $X_J = x_J$, we can assign a variable X_k to it ($k \notin J$) and add a set of children to this node equal to the number of values of X_k.
- *Pruning*–It is the inverse of branching. It consists in taking a node such that its children are leaves and removing the leaves and the variable assigned to it.

If $\mathcal{S}''$ is the structure obtained by any of these basic steps and ϕ' and ϕ'' are the potentials associated to trees $\mathcal{T}_{\mathcal{S}'}$ and $\mathcal{T}_{\mathcal{S}''}$ respectively, then branching is carried out trying to maximize $D(\phi'', \phi' \mid \psi)$ and pruning by trying to minimize $D(\phi', \phi'' \mid \psi)$. In both cases, it involves the computation of the Kullback-Leibler distance of a potential φ to another potential φ' given a third potential ψ. The value $\varphi'(x)$ is equal to $\varphi(x)$ in all the points of U_I, except for a subset $A \subseteq U_I$ in which $\varphi'(x) = \text{sum}(\varphi, A)/|A|$, where $|A|$ is the cardinal of A. In this case, the set A corresponds to all the values $x_I \in U_I$ such that following the path associated to it we arrive to the node before branching or after pruning, depending of the concrete operation we are carrying out. Making some easy calculations, this distance, $D(\varphi, \varphi' \mid \psi)$, can be calculated according to the following formula:

$$\frac{\left(\sum_{x \in A}(\varphi(x)\psi(x)\log(\varphi(x))) - \text{sum}(\varphi \cdot \psi|A)\log(\text{sum}(\varphi, A)/|A|)\right)}{\text{sum}(\varphi \cdot \psi)}$$

$$+ \log\left(\frac{\text{sum}(\varphi \cdot \psi) - \text{sum}(\varphi \cdot \psi|A) + (\text{sum}(\varphi|A)\text{sum}(\psi|A))/|A|}{\text{sum}(\varphi \cdot \psi)} \right)$$

We have used this expression to implement two procedures to simplify a tree $\mathcal{T}$ representing potential ϕ conditional on the potential represented by probability tree $\mathcal{T}_c$.

- **SortAndBound**($\mathcal{T}$,$\mathcal{T}_c$, *Size*, *Exact*)−This procedure returns an approximation tree $\mathcal{T}_a$ to the potential ϕ represented by $\mathcal{T}$ conditional on $\mathcal{T}_c$, starting with an empty tree and branching in each moment by the node given rise to a tree with greater distance to the current tree. The tree is built until a total of *Size* nodes is reached. There is a boolean variable *Exact* that is set to true when we have been able to build an exact representation of ϕ. It will be used to control the iterative improvements of messages by using more than two propagation stages in the junction tree as was indicated earlier.
- **Prune**($\mathcal{T}$,$\mathcal{T}_c$, ϵ_1, ϵ_2, *Exact*)−This procedure returns an approximation of the potential represented by $\mathcal{T}$ conditional on $\mathcal{T}_c$ starting with the structure of $\mathcal{T}$ and then carrying out several prunings of this structure. The condition for pruning the children of a node is that the distance to the resulting tree is less than ϵ_1. We have another value ϵ_2 that should be much lower than ϵ_1. If all the approximations have had a distance lower than ϵ_2, the boolean variable *Exact* is set to true. This variable is used similarly to the above one. The idea is that very small approximations (with a distance lower than ϵ_2) are not improved through successive iterations.

With respect to the complexity of these procedures, **Prune** is more efficient, since it uses the structure of tree $\mathcal{T}$. To calculate all the elements necessary for the conditional information, the algorithms needed to compute $\mathcal{T} \otimes \mathcal{T}_c$, that in the worst case has a complexity of the product of the sizes of the trees, but usually it is lower. Apart from this computation, **Prune** is linear in the size of $\mathcal{T}$. If the tree is not balanced **SortAndBound** has a complexity of $O(\log(\text{Size}) \cdot \text{Size} \cdot N)$, where N is the number of nodes of $\mathcal{T}$. The complexity expression, log(Size), comes from a priority queue that is maintained to select in each case the node with maximum distance. If $\mathcal{T}$ is balanced then this complexity is reduced to $O(\log^2(\text{Size}) \cdot N)$.

The approximations can be used in an isolated way or in combination. In that case **SortAndBound** is previous to **Prune**. If we set *Size* to a very high value, then **SortAndBound** builds a new structure for the tree but without actually doing an approximation. In that case, we will obtain an approximation by applying **Prune** afterwards.

5. EXPERIMENTS

To study the performance of penniless algorithm, we have carried out some experiments using three networks. The first one is a large network with 441 variables (`pedigree4`). This network is a subset of a pedigree one.[28] Each node except the roots has two parents and a maximum of 43 children. The set of possible cases for each node is 3. The second and the third ones are random networks with 60 and 100 variables respectively (`random60` and `random100`). The number of cases for each node is 2. The structure is quite complex and it is determined in the following way: each node has a number of parents determined by a Poisson distribution of $\lambda = 2$ and with a minimum of 1. The parents of a

node are randomly selected among the predecessors under a given ordering of nodes. This gives rise to a complex structure. The probabilities are determined in the following way: two random uniform numbers, x and y, are chosen from the interval $[0, 1]$; then the probabilities of both values of a variable are determined by normalizing x^5 and y^5. The result is that all the probabilities are very extreme: close to 0 or 1. This makes the propagation problem specially difficult for approximate algorithms.

We have done a number of experiments with these networks. We have considered the cases of observed and non-observed variables. When considering evidence, we have set 166 variables for `pedigree4`, 6 for `random60` and 10 for `random100`. In the random networks the observations are selected according to a uniform distribution. Also, we have carried out propagations with and without **SortAndBound** (Section 4.3). In both cases we use the procedure **Prune** described in Section 4.3. The trials have been carried out using different values of ϵ_1 for procedure **Prune** and different number of stages. We always have used $\epsilon_2 = 0.0$, i.e., the computations are considered exactly only in the case in which no approximation is carried out: the cross entropy is 0.

The results of penniless algorithm have been compared with exact results using the following measure of error[29] for each not observed variable X_l:

$$G(X_l) = \sqrt{\frac{1}{|U_l|} \sum_{a \in U_l} \frac{(\hat{p}(a \mid e) - p(a \mid e))^2}{p(a \mid e)(1 - p(a \mid e))}} \tag{10}$$

where $p(a \mid e)$ is the exact *posterior probability*, $\hat{p}(a \mid e)$ is the estimated value and $|U_l|$ is the number of cases of variable X_l. For a set of variables X_I, the error is:

$$G(X_I) = \sqrt{\sum_{i \in I} G(X_i)^2} \tag{11}$$

For each trial we have calculated the computing time and G-error (equation (11)). Penniless algorithm has been implemented in the Java language. Trials have been run on an Intel Pentium II (400 MHz) computer with 384MB of RAM and operating system Linux RedHat with kernel 2.0.36. The java virtual machine used is jdk version 1.2. In Table I we can see the computing time spent by an exact algorithm (HUGIN [13]) with these three networks. This algorithm has been also implemented in Java. In `random100` the Hugin algorithm was not possible to run (out of memory error) and then exact values were computed by

Table I. Time (in seconds) for pedigree4, random60 and random100 with Hugin algorithm.

	Without evidence	With evidence
pedigree4	2212.8	2339.7
random60	1167.4	1223.2
random100	????	????

repeating a deletion algorithm for each variable. The time was much higher, but we could use a reduced network for each variable so that the computations were possible.

In Tables II to IV we can see the error and computing time for each network and each trial with and without **SortAndBound** (**SaB** or **No SaB** in table) and with different number of stages and ϵ_1.

5.1. Results Discussion

The first conclusion is that very good approximations are possible in a very short time. For example, in the pedigree network with observations, in a few seconds we can have very good approximations (in the best experiments, these errors imply that approximate probabilities are really close to exact probabilities, being the maximum absolute difference between exact and approximate probabilities of 0.0006, and in most of the cases the exact probability value is found up to the 7th decimal digit). The time of our exact algorithm was more than half an hour. In the random networks the approximations are also very good when **SortAndBound** is not applied and with the smallest value of ϵ_1, but we have to spend more time. In the case of observations, we obtain absolute differences always lower than 0.0005 in `random60` and lower than 0.0006 `random100`. In the pedigree example, good approximations are obtained also for small values of ϵ_1 which imply less time. Anyway, we have to take into account that the random problems are artificially difficult and that, in any case, the times are quite low compared with exact algorithms.

The second conclusion is that the application of **SortAndBound** is not reasonable. We never have obtained a smaller error in a time equal or lower than without applying it. It is even the case than using the same parameters (number of runs and ϵ_1) applying **SortAndBound** we have obtained a greater error (see Figures 13 to 18). This can be seen as counterintuitive as this is a technique commonly used to approximate decision trees. Our reason for this fact is the following: in general we have a potential ϕ which is represented by a tree $\mathcal{T}$ and we look for an optimal approximation $\mathcal{T}'$. With **SortAndBound** we forget the structure of $\mathcal{T}$, and very often the number of values of $\mathcal{T}$ is very small compared with the number of values necessary to completely represent ϕ by means of a table. As the procedure used in **SortAndBound** is only an heuristics and not a globally optimal procedure, it is possible that we obtain a worst representation than the one provided by $\mathcal{T}$, which already was very good. However, we do not feel that we could extrapolate this conclusion to every possible situation. There can be cases in which rebuilding a tree can be a good idea. For example, in the pedigree example without evidence (see Fig. 13), experiments without **SortandBound** are better than those with it. Probably, the best solution would not be in the application of **SortAndBound**, but in some other intermediate procedures trying to find good structures for tree construction. For example, now in the implementation of combination and addition of trees, the structure of one tree is taken as basis and the other is built on top of it by branching in its leaves. The algorithms could mix the structures of both trees

Table II. Error and time (in seconds) for pedigree4 with penniless algorithm.

		Without Evidence			With Evidence		
Stages	ϵ_1	No SaB	SaB Tree:50	SaB Tree:100	No SaB	SaB Tree:50	SaB Tree:100
2	10^{-3}	0.0872—0.0468	0.00448—0.0	0.0872—0.0468	0.00448—0.0	0.0872—0.0468	0.00448—0.0
	10^{-4}	0.195—174.1	0.355—27.8	0.207—45.8	0.0191—5.5	0.383—9.1	0.0338—7.6
	10^{-5}	0.199—494.8	0.303—32.7	0.209—53.4	0.00376—5.1	0.396—8.7	0.205—7.8
	10^{-6}	0.199—902.3	0.304—33.9	0.187—55.7	0.00179—5.3	0.396—8.5	0.338—7.8
4	10^{-3}	0.205—54.3	0.304—27.8	0.223—36.3	0.140—5.6	0.232—8.5	0.169—7.7
	10^{-4}	0.195—174.9	0.266—37.9	0.205—54.2	0.0294—5.6	0.365—9.4	0.0335—8.2
	10^{-5}	0.199—512.2	0.298—46.8	0.227—65.7	0.00475—5.6	0.378—9.7	0.197—8.6
	10^{-6}	0.199—903.9	0.312—47.4	0.208—72.3	0.00179—6.0	0.412—9.5	0.199—8.5
6	10^{-3}	0.205—55.7	0.309—32.5	0.215—47.5	0.140—5.6	0.223—8.8	0.149—8.4
	10^{-4}	0.195—175.3	0.266—47.5	0.211—61.3	0.0294—5.8	0.365—10.0	0.0335—8.6
	10^{-5}	0.199—512.5	0.340—57.9	0.206—81.7	0.00475—6.7	0.458—11.1	0.197—8.8
	10^{-6}	0.199—905.0	0.325—56.7	0.208—81.4	0.00179—6.6	0.435—12.3	0.199—9.0

Table III. Error and time (in seconds) for random60 with penniless algorithm.

		Without Evidence			With Evidence		
Stages	ϵ_1	No SaB	SaB Tree:1000	SaB Tree:5000	No SaB	SaB Tree:1000	SaB Tree:5000
2	10^{-2}	0.839—26.4	0.915—53.6	0.807—42.1	0.476—20.5	5.01—30.9	5.01—32.0
	10^{-3}	0.120—49.9	0.396—86.9	0.162—97.9	0.111—42.2	17.7—62.4	1.11—62.3
	10^{-4}	0.0392—95.5	0.470—132.7	0.0429—183.5	0.0174—56.7	8.06—92.2	0.163—95.6
	10^{-5}	0.0344—117.3	0.490—176.5	0.338—266.7	0.00338—71.9	47.5—118.7	0.00844—141.2
4	10^{-2}	0.853—47.5	0.731—72.3	0.807—72.5	0.317—35.4	3.94—54.3	3.94—54.1
	10^{-3}	0.142—78.8	0.387—158.1	0.170—191.3	0.0518—71.2	1.65—97.7	0.223—112.8
	10^{-4}	0.0391—138.9	0.542—224.6	0.0430—303.3	0.0162—115.6	27.5—173.1	0.0534—181.7
	10^{-5}	0.0344—185.4	0.462—281.0	0.238—505.6	0.00330—119.0	92.3—201.8	0.00785—260.7
6	10^{-2}	0.852—61.8	0.717—100.5	0.821—106.2	0.317—44.1	1.52—70.6	1.52—70.3
	10^{-3}	0.143—109.9	0.479—249.1	0.155—263.5	0.0515—79.2	20.2—145.2	0.188—150.0
	10^{-4}	0.0391—164.9	0.540—329.4	0.0421—414.5	0.0159—104.4	3.21—238.9	0.0403—251.7
	10^{-5}	0.0344—261.8	0.640—376.5	0.390—627.9	0.00331—140.5	83.5—288.0	0.00802—355.6

Table IV. Error and time (in seconds) for random100 with penniless algorithm.

		Without Evidence			With Evidence		
Stages	ϵ_1	No SaB	SaB Tree:1000	SaB Tree:5000	No SaB	SaB Tree:1000	SaB Tree:5000
2	10^{-2}	1.31—102.4	1.17—117.1	1.16—108.0	1.75—26.9	6.20—38.2	6.20—33.4
	10^{-3}	0.194—314.7	1.73—251.4	0.240—360.3	0.303—52.7	1.11—75.4	1.079—67.4
	10^{-4}	0.0402—869.1	3.03—434.0	0.704—849.9	0.061—76.1	0.219—114.7	0.103—118.3
	10^{-5}		3.50—607.5	0.809—1466.3	0.00722—120.6	0.970—180.8	0.0230—185.2
4	10^{-2}	0.886—143.5	1.05—174.6	1.05—166.2	4.12—31.0	2.04—61.2	2.02—63.5
	10^{-3}	0.160—375.8	2.42—464.6	0.195—599.4	0.282—60.5	0.668—109.5	0.786—112.8
	10^{-4}	0.0367—1032.4	2.94—729.1	0.697—1507.0	0.0681—92.4	0.714—167.3	0.139—168.3
	10^{-5}		3.61—1156.8	1.93—2433.7	0.00812—134.0	0.762—263.5	0.0623—257.3
6	10^{-2}	0.871—167.0	1.33—244.3	1.17—276.2	2.468—39.3	1.89—85.9	2.91—85.3
	10^{-3}	0.161—479.4	2.74—628.6	0.200—903.2	0.279—80.6	0.601—145.2	0.684—152.0
	10^{-4}	0.0371—1241.6	3.48—1037.3	0.713—2005.1	0.0677—110.0	0.410—218.2	0.141—226.3
	10^{-5}		3.53—1594.6	2.10—3132.4	0.00773—175.5	0.657—345.9	0.0819—330.1

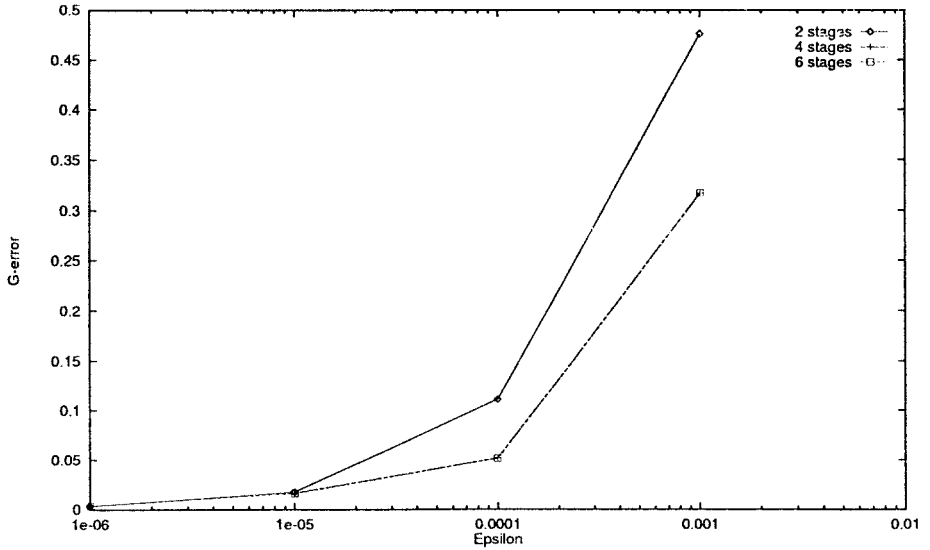

Figure 11. G-error for each ϵ_1 (log scale) in random60 with evidence without **SortAndBound**.

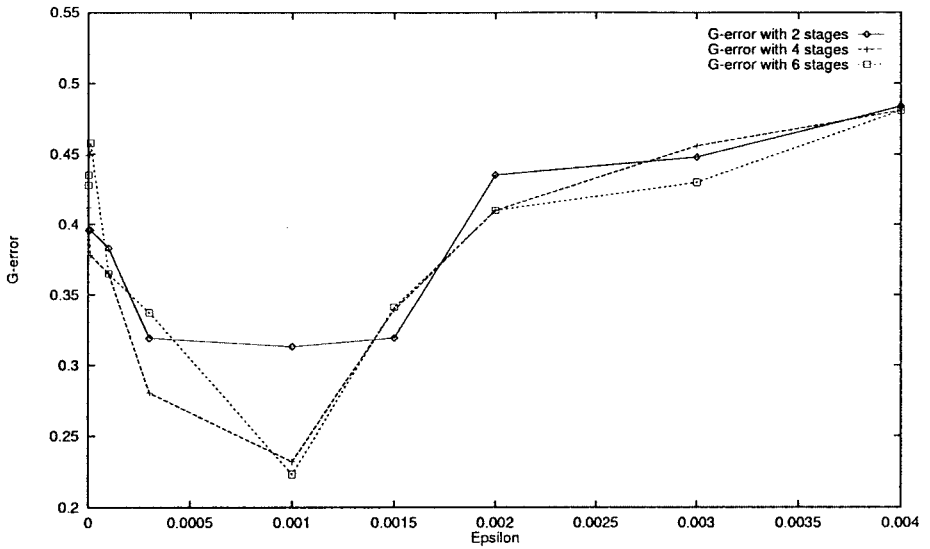

Figure 12. G-error for each ϵ_1 in pedigree4 with evidence with **SortandBound** (Tree size: 50).

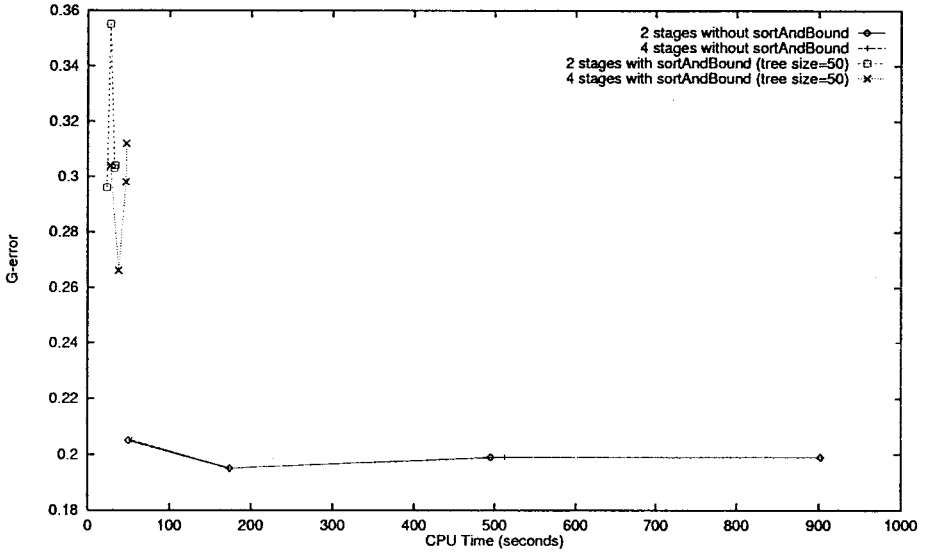

Figure 13. G-error in pedigree4 without evidence.

if the selection of the root of the result is not always chosen from one of them (in our case, if the first tree has a variable on its root then this will be the root of the result). An heuristics could be used to select informative roots from the roots of the operators. This could be a possible alternative to improve the representation of potentials without a big additional cost on time.

In general, decreasing the value of ϵ_1 decreases the value of the error, except for very small values when applying **SortAndBound**. Figures 11 and 12

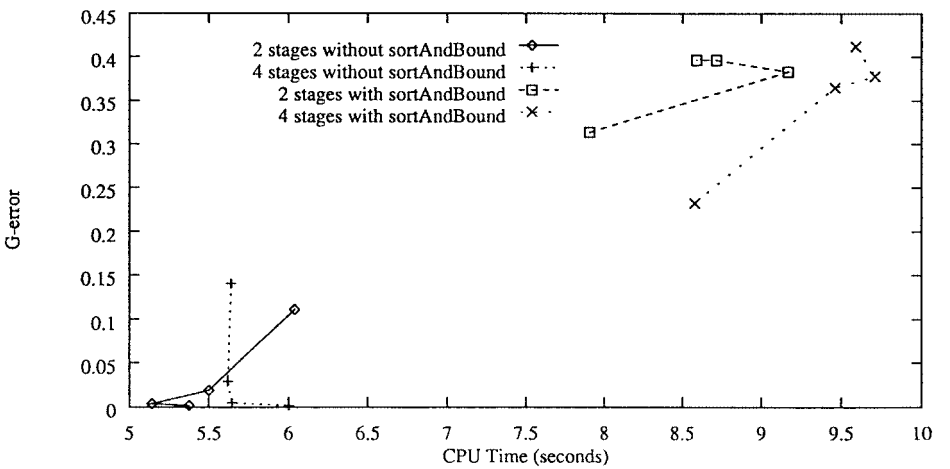

Figure 14. G-error in pedigree4 with evidence.

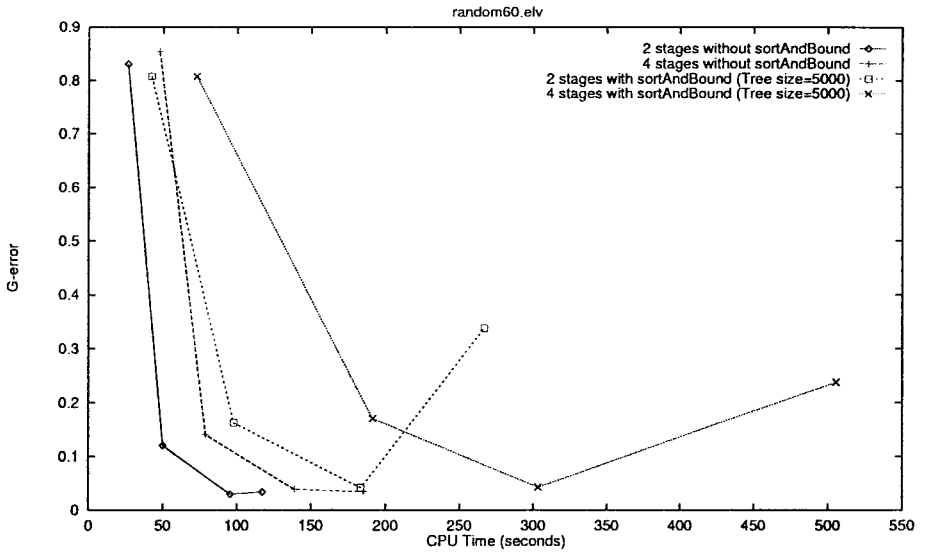

Figure 15. G-error in random60 without evidence.

represent typical cases of these situations. The explanation of the increasing of the error with **SortAndBound** (see Fig. 12) is that doing less approximations with **Prune** makes more difficult finding a new structure for the approximate tree, and the fact that the initial tree is more exact does not compensate for the increasing in the complexity of the problem to solve.

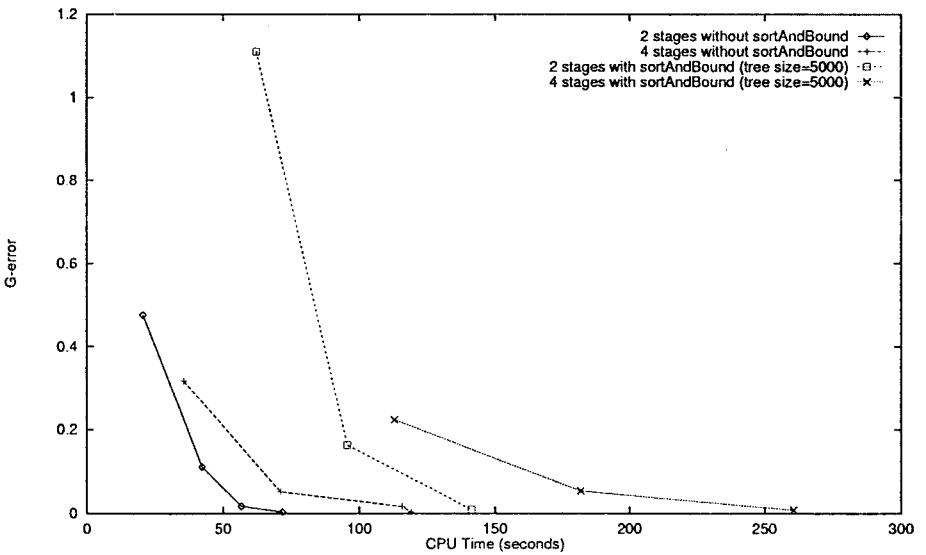

Figure 16. G-error in random60 with evidence.

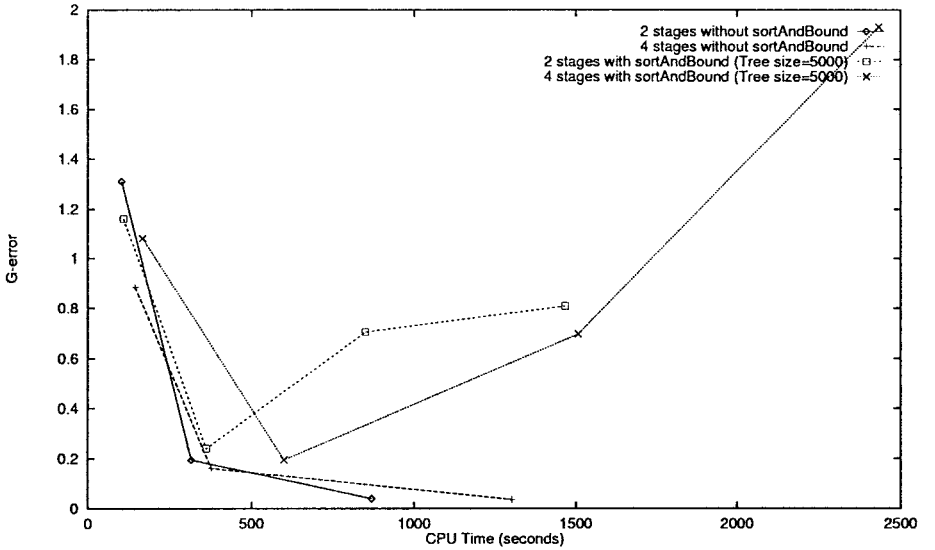

Figure 17. G-error in random100 without evidence.

Another important fact is that, in most of the cases, carrying out more runs decrease the error. There are cases in which this is not the case, and in fact we do expect that it could be proved that in the limit we should always obtain a better approximation. What really happens is that the error is stabilized in a few runs and we feel that a theoretical result in this direction could be proved. A

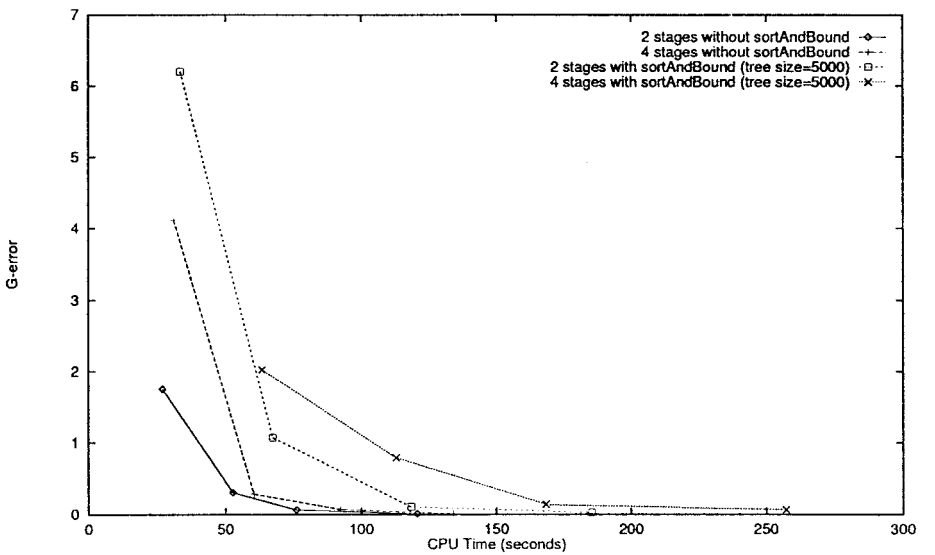

Figure 18. G-error in random100 with evidence.

typical situation is shown in Figure 11. Another comment relative to the use of multiple runs, is that the time spent is not proportional to the number of runs. This fact is due to our implementation in which no additional runs are carried out when one of the messages is exact. This fact is more evident when there are no observations. As our triangulation is carried out in such a way that non-observed leaves in the original Bayesian network are deleted first, this gives rise to upward messages such that most of them are equal to 1, and then they can be approximate without problems. This fact is not fully exploited in our implementation as making approximations before the calculation of this message may give rise to messages that are not 1. In the following section we will give some idea of how to do it in the future. Though making 4 runs does not imply the double of time than 2 runs, in general without applying **SortAndBound** the decreasing in the error does not compensate for the increment of time. Our opinion however is that it is a good idea to carry out more than 2 runs: to continue until the results are stable. We think that this will be a security for difficult odd situations and the payment in time is not great. We could even reduce the extra time by considering higher values for ϵ_2, with which computations are only repeated in some parts of the network.

Another fact to take into account is that the use of conditional information is limited if we keep fix the structures of the approximation trees (without **SortAndBound**) but it could be more important in the future if we incorporate the conditional information to mix the structures of the trees when we are performing operations over them. Another point to consider is the possibility of using different values of ϵ_1 for different runs: for example we could consider bigger errors in the first runs used to guide the last ones, which are carried out with smaller values of ϵ_1.

6. FUTURE WORK

In this paper we have presented an approximation scheme for probability propagation in Bayesian networks. Though the results are quite good, we think that this is only one step and there is room for further improvements on the algorithms. Some of them have been pointed out in the discussion section. One of the most promising possibilities from our point of view is the use of the lazy propagation technique.[30,5] This is important because it will decrease the number of operations and approximations to be carried out. In a binary join tree there are additional cluster nodes which do not involve marginalizations to compute the messages. With lazy propagation these messages can be computed by copying references to potential instead of deep copies which are more expensive on time. We could keep original conditional probabilities as potentials such that when adding on one of its variables the result is the potential identically equal to 1 without making any computation, thus the computation of easy messages (almost half of them in the case of no observations) is extremely fast.

Another feature of these algorithms is the possibility of computing an interval for the final probabilities. Each time we make an approximation of a set of leaves by a node we can compute the maximum and minimum values of the

leaves, which can be used to compute intervals for the conditional probability values as in Ref. 16. If the aim is to optimize intervals (produce intervals as small as possible) then the measure of information could be different. This fact will be investigated in the future.

References

1. Jensen FV, Lauritzen SL, Olesen KG. Bayesian updating in causal probabilistic networks by local computation. Comput Stat Quarterly 1990;4:269–282.
2. Lauritzen SL, Spiegelhalter DJ. Local computations with probabilities on graphical structures and their application to expert systems. J Roy Stat Soc Ser B 1988;50: 157–224.
3. Shenoy PP. Binary join trees for computing marginals in the Shenoy-Shafer architecture. Int J Approx Reas 1997;17:239–263.
4. Shenoy PP, Shafer G. Axioms for probability and belief function propagation. In: Shachter RD, Levitt TS, Lemmer JF, Kanal LN, editors. Uncertainty in artificial intelligence 4. Amsterdam: North Holland; 1990. p 169–198.
5. Madsen AL, Jensen FV. Lazy propagation: a junction tree inference algorithm based on lazy evaluation. Artif Intell 1999;113:203–245.
6. Bouckaert RR, Castillo E, Gutiérrez JM. A modified simulation scheme for inference in Bayesian networks. Int J Approx Reas 1996;14:55–80.
7. Cano JE, Hernández LD, Moral S. Importance sampling algorithms for the propagation of probabilities in belief networks. Int J Approx Reas 1996;15:77–92.
8. Dagum P, Luby M. An optimal approximation algorithm for Bayesian inference. Artif Intell 1997;93:1–27.
9. Fung R, Chang KC. Weighting and integrating evidence for stochastic simulation in Bayesian networks. In: Henrion M, Shachter RD, Kanal LN, Lemmer JF, editors. Uncertainty in artificial intelligence 5. Amsterdam: North-Holland; 1990. p 209–220.
10. Salmerón A, Cano A, Moral S. Importance sampling in Bayesian networks using probability trees. Comput Statis Data Analysis 2000. To appear.
11. Shachter RD, Peot MA. Simulation approaches to general probabilistic inference on belief networks. In: Henrion M, Shachter RD, Kanal LN, Lemmer JF, editors. Uncertainty in artificial intelligence 5. Amsterdam: North Holland; 1990. p 221–231.
12. Hernández LD, Mora S, Salmerón A. A monte carlo algorithm for probabilistic propagation in belief networks based on importance sampling and stratified simulation techniques. Int J Approx Reas 1998;18:53–91.
13. Jensen F, Andersen SK. Approximations in Bayesian belief universes for knowledge-based systems. In: Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence; 1990. p 162–169.
14. Kjaerulff U. Reduction of computational complexity in Bayesian networks through removal of weak dependencies. In: Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence. San Francisco: Morgan Kaufmann; 1994. p 374–382.
15. Poole D. Average-case analysis of a search algorithm for estimating prior and posterior probabilities in Bayesian networks with extreme probabilities. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93). San Mateo: Morgan Kaufmann; 1993. p 606–612.
16. Santos E, Shimony SE. Belief updating by enumerating high-probability independence-based assignments. In: Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence; 1994. p 506–513.
17. Santos E, Shimony SE, Williams E. Hybrid algorithms for approximate belief updating in bayes nets. Int J Approx Reas 1997;17:191–216.
18. Pearl J. Probabilistic reasoning in intelligent systems. San Mateo: Morgan Kaufmann; 1988.

19. Jensen FV. An introduction to Bayesian networks. UCL Press; 1996.
20. Cano A, Moral S. Heuristic algorithms for the triangulation of graphs. In: Bouchon-Meunier B, Yager RR, Zadeh L, editors. Advances in intelligent computing. Springer-Verlag; 1995. p 98–107.
21. Kjaerulff U. Optimal decomposition of probabilistic networks by simulated annealing. Statis Comput 1992;2:1–21.
22. Boutilier J, Friedman N, Goldszmidt M, Koller D. Context-specific independence in Bayesian networks. In: Horvitz E, Jensen FV, editors. Proceedings of the 12th conference on uncertainty in artificial intelligence. Morgan Kaufmann; 1996. p 115–123.
23. Kozlov D, Koller D. Nonuniform dynamic discretization in hybrid networks. In: Geiger D, Shenoy PP, editors. Proceedings of the 13th conference on uncertainty in artificial intelligence. Morgan Kaufmann; 1997. p 302–313.
24. Kozlov AV. Efficient inference in Bayesian networks, Ph.D. thesis. Stanford University; 1998.
25. Cano A, Moral S, Propagación exacta y aproximada con árboles de probabilidad. In: Actas de la VII Conferencia de la Associación Española para la Inteligencia Artificial; 1997. p 635–644.
26. Koller D, Lerner U, Anguelov D5. A general algorithm for approximate inference and its application to hybrid Bayes nets. In: Laskey KB, Prade H, editors. Proceedings of the 15th conference on uncertainty in artificial intelligence. Morgan Kaufmann; 1999. p 324–333.
27. Kullback S, Leibler R. On information and sufficiency. Annals Mathematical Statis 1951;22:76–86.
28. Jensen CS, Kong A, Kjaerulff U. Blocking gibbs sampling in very large probabilistic expert systems. Int J Human-Computer Studies 1995;42:647–666.
29. Fertig KW, Mann NR. An accurate approximation to the sampling distribution of the studentized extreme-valued statistic. Technometrics 1980;22:83–90.
30. Madsen AL, Jensen FV. Lazy propagation in junction trees. In: Cooper GF, Moral S, editors. Proceedings of the 14th conference on uncertainty in artificial intelligence 113. Morgan Kaufmann; 1998. p 362–369.