Simulation of Graphical Models for Multiagent Probabilistic Inference

Y. Xiang
Department of Computing and Information Science
College of Physical and Engineering Science
University of Guelph
Guelph, Ontario
Canada N1G 2W1
yxiang@cis.uoguelph.ca

X. An
School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
xan@cs.waterloo.ca

N. Cercone
Faculty of Computer Science
Dalhousie University
6050 University Avenue
Halifax, Nova Scotia Canada B3H 1W5

Multiply-sectioned Bayesian networks (MSBNs) extend Bayesian networks to graphical models for multiagent probabilistic reasoning. The empirical study of algorithms for manipulations of MSBNs (e.g., verification, compilation, and inference) requires experimental MSBNs. As engineering MSBNs in large problem domains requires significant knowledge and engineering effort, the authors explore automatic simulation of MSBNs. Due to the large domain over which an MSBN is defined and a set of constraints to be satisfied, a generate-and-test approach toward simulation has a high rate of failure. The authors present an alternative approach that treats the simulation process as a sequence of decisions. They constrain the space of each decision so that backtracking is minimized and the outcome is always a legal MSBN. A suite of algorithms that implements this approach is presented, and experimental results are shown.

Keywords: Simulation algorithms, graphical models, multiagent systems, probabilistic reasoning, computer science

1. Introduction

As intelligent systems are being applied to larger, open, and more complex problem domains, many applications are found to be more suitably addressed by multiagent systems [1, 2]. More specifically, an application system in such a problem domain cannot be effectively constructed as a single intelligent agent due to the scale and complexity of the domain. Instead, a set of cooperating agents (each of them is much simpler) can be more effectively developed, each of which is embedded with a subset of knowledge and problem-solving abilities and pursues subtasks autonomously. Key characteristics of such a multiagent system include the following: each agent has incomplete knowledge or capabilities for solving the problem. There is no global control on problem-solving activities. Data are distributed. Computation is asynchronous.

Consider a large uncertain problem domain populated by a set of agents. The agents are often charged with many tasks determined by the nature of the application. One common task of the agents is to estimate what is the true state of the domain so that they can act accordingly.

SIMULATION, Vol. 79, Issue 10, October 2003 545-567

DOI: 10.1177/0037549703039950

Such a task, often referred to as *distributed interpretation* [3], arises in many applications of multiagent systems, including troubleshooting of a complex equipment/process, building/area surveillance, battlefield/disaster situation assessment, and distributed design. We can describe the domain with a set of variables. Some variables are not directly observable, and their values can only be inferred based on observations of other variables and the background knowledge on their dependence relations.

Furthermore, each agent has only a partial perspective of the problem domain. That is, each agent only has knowledge about a subdomain (i.e., about the dependence among a subset of domain variables) and can only observe and reason within the subdomain.

In the case of a single agent, the task of estimating the state of the domain can be achieved by representing the domain knowledge in a Bayesian network (BN) [4] and by performing probabilistic inference using the BN given the agent's observations. Multiply-sectioned Bayesian networks (MSBNs) [5] extend single-agent BNs to multiagent uncertain reasoning. An MSBN consists of a set of interrelated Bayesian subnets, each of which encodes an agent's knowledge on a subdomain. Agents are organized into a hypertree structure such that inference can be performed in a distributed fashion while answers to queries are exact with respect to probability theory. Each agent only exchanges information with adjacent agents on the hypertree, and each pair of adjacent agents only exchanges information on a set of shared variables. The complexity of communication among all agents is linear on the number of agents, and the complexity of local inference is the same as if the subnet were a single-agent-based BN. Therefore, MSBNs provide a framework in which multiple agents can estimate the state of a domain effectively with exact and distributed probabilistic inference.

As a small example, Figure 1 shows digital equipment made out of five components, U_i ($i = 0, ..., 4$). Each box in the figure corresponds to a component and contains the logical gates and their connections, with the input/output signals of each gate labeled. A set of five agents, A_i ($i = 0, ..., 4$), cooperates to monitor the equipment and troubleshoot it when necessary. Each agent A_i is responsible for a particular component, U_i, that is likely developed by the vendor of the component. When a gate is enclosed in exactly one box, the gate is *physically* located in the corresponding component and is *logically* known only to the agent responsible for the component. On the other hand, when a gate is enclosed in more than one box, the gate is *physically* located in only one of the components but is *logically* known to all the corresponding agents. For example, the AND gate g_5 is known only to A_1, the OR gate g_8 is known to both A_1 and A_2, and the signal z_2 is known to A_0, A_1, and A_2. The knowledge of an agent on its assigned component can be represented as a BN, called a *subnet*.

The subnet for agent A_1 (responsible for component U_1) is shown in Figure 2, and that for A_2 is shown in Figure 3. Each node is labeled with a variable name followed by an index. Only the directed acyclic graph (DAG) of each subnet is shown in the figure, with the conditional probability distribution for each variable omitted. Note that each subnet contains the knowledge of the agent about the physical devices (gates) of the component, as well as the knowledge about the devices that are only logically known to the agent. The five subnets (one for each component) collectively define an MSBN, which form the core knowledge representation of the multiagent system.

Based on this knowledge and limited observations, agents can cooperate to estimate whether the equipment is functioning normally and, if not, which devices are likely to be responsible. For instance, suppose that the gates d_1 and t_5 in Figure 1 break down and produce incorrect output. Some outputs downstream are also affected. Equipment inputs and correct device outputs are shown in Figure 1 by 0 and 1. Incorrect outputs are underlined. Through limited local observation and communication, agents can identify the two faulty gates [6] correctly.

To experimentally investigate algorithmic issues involved in the structural verification (whether a set of independently developed agents forms a correct MSBN), compilation (converting an MSBN into a more effective representation for inference computation), inference (how to perform inference more efficiently when the problem domain becomes very large), and other operations, a number of MSBNs of different structures and parameters are needed. Since MSBNs are intended for very large domains, manual construction of MSBNs of such scales is impractical.

We study random simulation of large MSBNs to fuel experimental study of MSBN-related algorithms. One obvious approach to simulation is generate-and- test (to construct randomly a graphical model and to test if the result is acceptable). However, due to the large number of variables to be simulated and multiple technical conditions (detailed in section 3) to be satisfied simultaneously, such an approach has a very low probability to produce acceptable MSBNs. It is also difficult to control the topological characteristics of the resultant MSBNs.

Alternatively, the simulation can be constrained so that "correct" decisions are made at each simulation step and the result of the simulation is always acceptable. In this work, we present a suite of algorithms that implements this approach.

Simulation of centralized (single-agent oriented) probabilistic graphical models has been used by many for experimental study of algorithms related to single-agent probabilistic reasoning. Very few of these algorithms (e.g., [7, 8]) were published. Closely related work includes methods for the automatic construction of such models (e.g., [9-17]). The objective is to dynamically construct a model after a new case is presented rather than to produce a large set of graphical models for experimental study. To the best of our knowledge, simulation of decentralized probabilis-

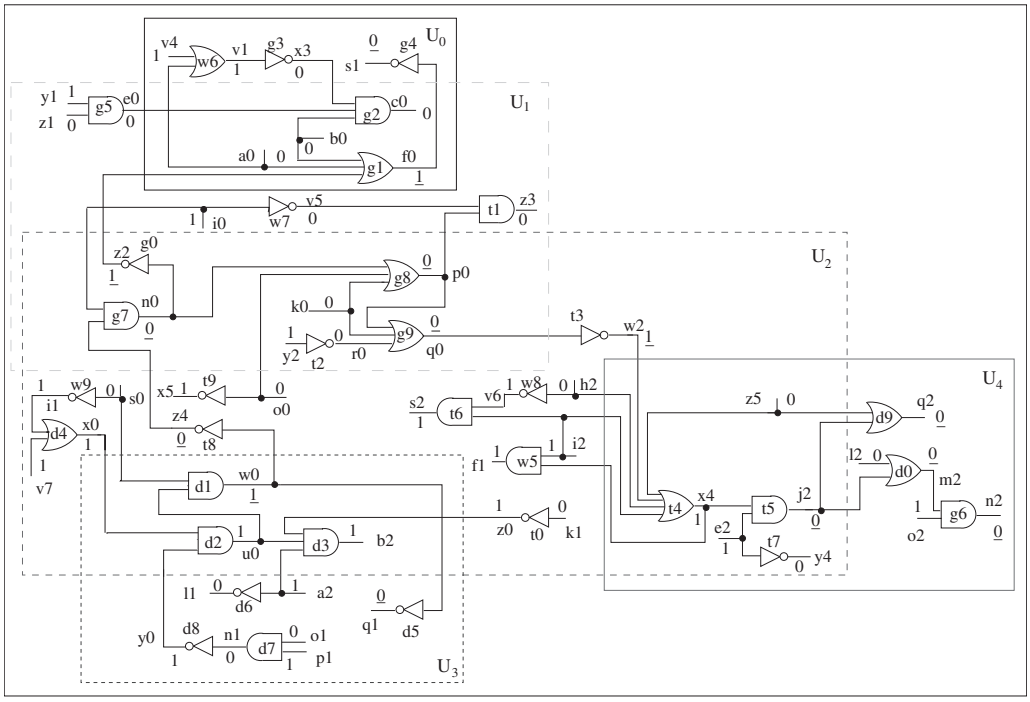

Figure 1. An example digital equipment

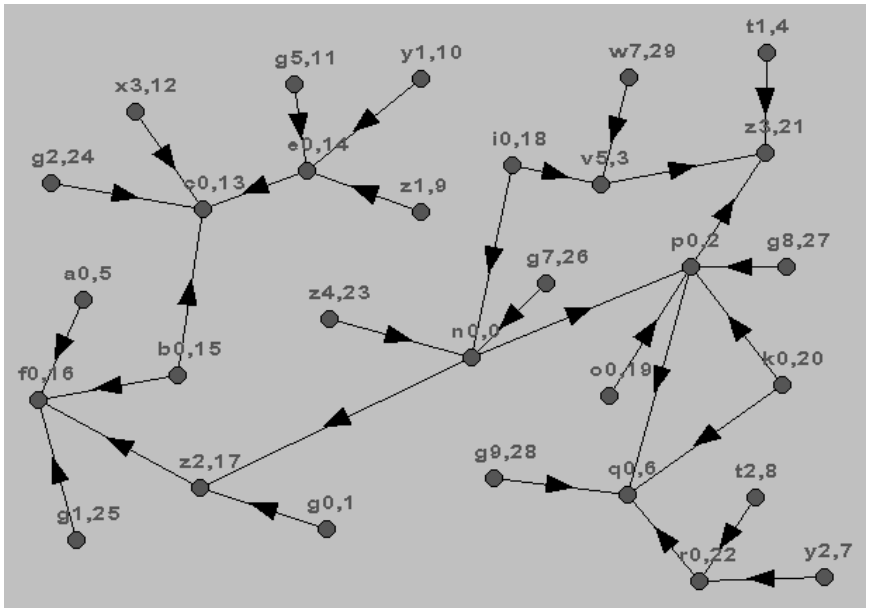

Figure 2. The subnet G_1 for U_1

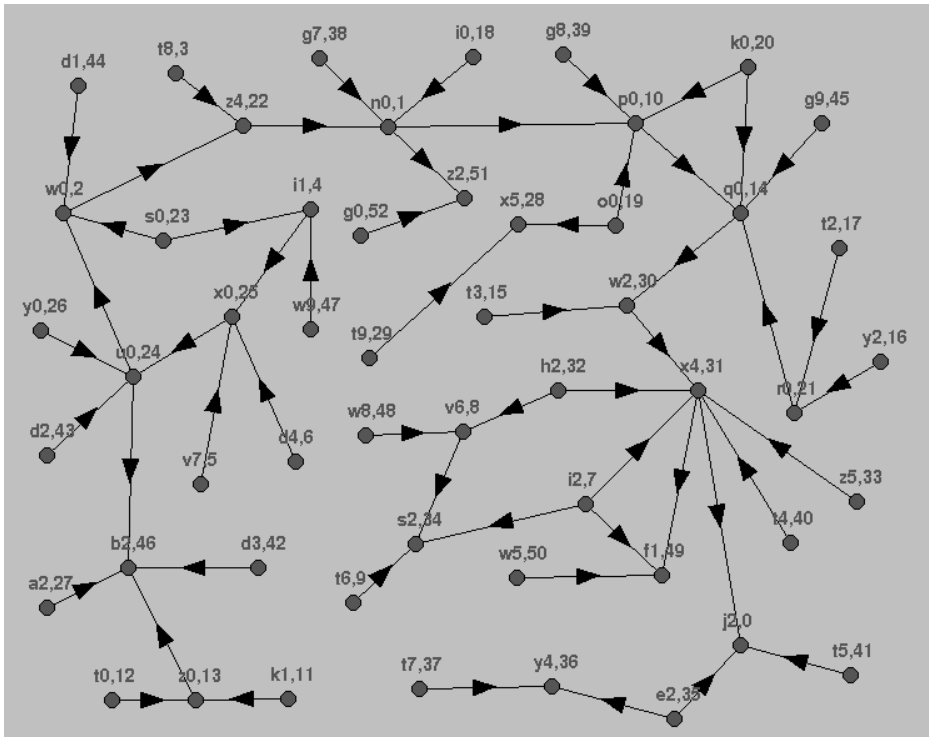

Figure 3. The subnet G_2 for U_2

tic graphical models has never been investigated. Hence, this work provides the first study of the issues involved and presents the first set of such algorithms.

The rest of the article is organized as follows: we overview the formal properties of the MSBN representation in section 2. In section 3, we specify a set of high-level criteria that guides our construction of the simulation algorithms. Toward the end of the section, we describe the major steps in simulating an MSBN. With these background introduced, we end the section by indicating how the remaining sections correspond to these steps.

2. Overview of MSBNs

For multiple agents to perform probabilistic inference distributively and exactly, an MSBN must satisfy a set of technical conditions. These conditions constitute the minimum set of constraints for the simulation of an MSBN. We introduce these conditions in this section and present additional constraints in section 3.

A BN [4] S is a triplet (V, G, P), where V is a set of domain variables, G is a connected DAG whose nodes are labeled by elements of V (hence we refer to variables and nodes interchangeably), and P is a joint probability distribution (jpd) over V. An MSBN [5] M is a collection of Bayesian subnets that together defines a BN. Each subnet corresponds to an agent (the subnet forms the core knowledge of the agent). For agents to reason exactly through message passing, their communication pathways need to satisfy a condition known as a *hypertree*, defined as follows.

Let $G_i = (V_i, E_i)$ $(i = 0, 1, ..., n-1)$ be a set of graphs where V_i is the set of nodes and E_i is the set of edges in graph G_i. The graph $G = (\bigcup_i V_i, \bigcup_i E_i)$, also denoted by $\bigsqcup G_i$, is referred to as the *union* of G_0, G_1,..., and G_{n-1}.

DEFINITION 1. Let $G = (V, E)$ be a connected graph sectioned into subgraphs $\{G_i = (V_i, E_i)\}$. Let $G_i s$ be organized as a connected tree Ψ, where each node is labeled by a G_i and each link between G_k and G_m is labeled by the `interface` $V_k \cap V_m$ such that for each i and j, $V_i \cap V_j$ is contained in each subgraph on the path between G_i and G_j in Ψ. Then Ψ is a `hypertree` over G. Each G_i is a `hypernode`, and each interface is a `hyperlink`.

Figure 4 illustrates a hypertree for the digital equipment. Another condition that acts together with the hypertree condition to ensure the correctness of agent communication is that nodes shared by agents should form a *d-sepset*:

DEFINITION 2. Let G be a directed graph such that a hypertree over G exists. A node x contained in more than one subgraph with its parents $\pi(x)$ in G is a `d-sepnode` if there exists one subgraph that contains $\pi(x)$. An interface I is a `d-sepset` if every $x \in I$ is a `d-sepnode`.

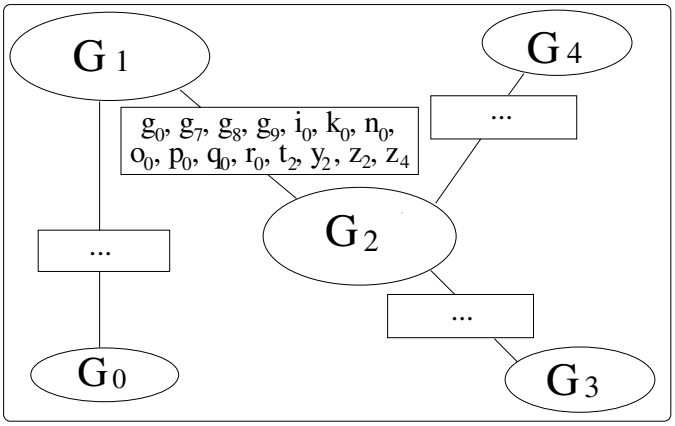

Figure 4. The hypertree for the digital equipment monitoring system

The interface between G_1 (Fig. 3, top) and G_2 (Fig. 3, bottom) contains 13 variables indicated in Figure 4. It is a d-sepset since these variables are only shared by G_1 and G_2, and each variable has all its parents contained in one of them. For instance, the parents of z_4 are all contained in G_2, while those of n_0 are contained in both G_1 and G_2.

The structure of an MSBN is a multiply-sectioned DAG (MSDAG) with a hypertree organization:

DEFINITION 3. A `hypertree MSDAG` $G = \bigsqcup_i G_i$, where each G_i is a connected DAG, is a connected DAG such that (1) there exists a hypertree Ψ over G, and (2) each hyperlink in Ψ is a d-sepset.

An MSBN consists of its structure as defined above as well as its numerical probability distributions. The following definition of an MSBN specifies how the numerical distributions are associated with the structure:

DEFINITION 4. An MSBN M is a triplet (V, G, P). $V = \bigcup_i V_i$ is the `total universe`, where each V_i is a set of variables called a `subdomain`. $G = \bigsqcup_i G_i$ (a hypertree MSDAG) is the `structure` where nodes of each subgraph G_i are labeled by elements of V_i. Let x be a variable and $\pi(x)$ be all parents of x in G. For each x, exactly one of its occurrences (in a G_i containing $\{x\} \cup \pi(x)$) is assigned $P(x|\pi(x))$, and each occurrence in other subgraphs is assigned a unit constant potential. $P = \prod_i P_i$ is the `jpd`, where each P_i is the product of the potentials associated with nodes in G_i. Each triplet $S_i = (V_i, G_i, P_i)$ is called a `subnet` of M. Two subnets S_i and S_j are said to be adjacent if G_i and G_j are adjacent in the hypertree.

More details on MSBNs and its application to multiagent probabilistic reasoning can be found in Xiang [18].

3. Criteria for Simulation

The minimum set of conditions that a simulated graphical model must satisfy is outlined in section 2. To summarize, the structure of a *legal* MSBN should have the following properties (we use the terms *hypernode* and *subnet* interchangeably):

- A hypertree organization is used for the subnets.
- Each subnet is a connected DAG.
- The union of all subnets is also a connected DAG.
- Nodes shared by adjacent subnets form a d-sepset.

Although it seems that the local probability distributions can be arbitrarily simulated subject to Definition 4, they should be consistent with the graphical connections. For instance, if a node y is the sole parent of a node x, it is intended to convey the dependence of x on y. Hence, if a distribution $P(x|y)$ is simulated, it should not be the case that $P(x|y) = P(x)$. This requirement can be generalized to the case with multiple parent nodes.

In addition to the above legal requirement, ideally, it should be possible for the simulation process to produce MSBNs of all legal structures and distributions. In other words, all legal MSBNs should have nonzero probabilities (equal probabilities would be even better) to be simulated (if we allow our control parameters to be evenly distributed). This is a *reachability* criterion. Even if one has a set of simulation algorithms that satisfies the criterion, it appears to be quite difficult to prove so formally. Instead, we empirically evaluate our simulation algorithms against the reachability and require that the simulated MSBNs demonstrate a variety of characteristics. For instance, when a pair of adjacent subnets shares a set I of d-sepnodes, elements of I may or may not be shared by other subnets. Simulated MSBNs should display both types of topologies.

Using an MSBN, multiple agents can perform probabilistic reasoning by local inference and interagent message passing. To make the local inference efficient, each subnet must be sparse. To make message passing efficient, the d-sepset between each pair of adjacent agents (whose subnets are adjacent in the hypertree) should be small compared with the corresponding subdomains. The d-sepset

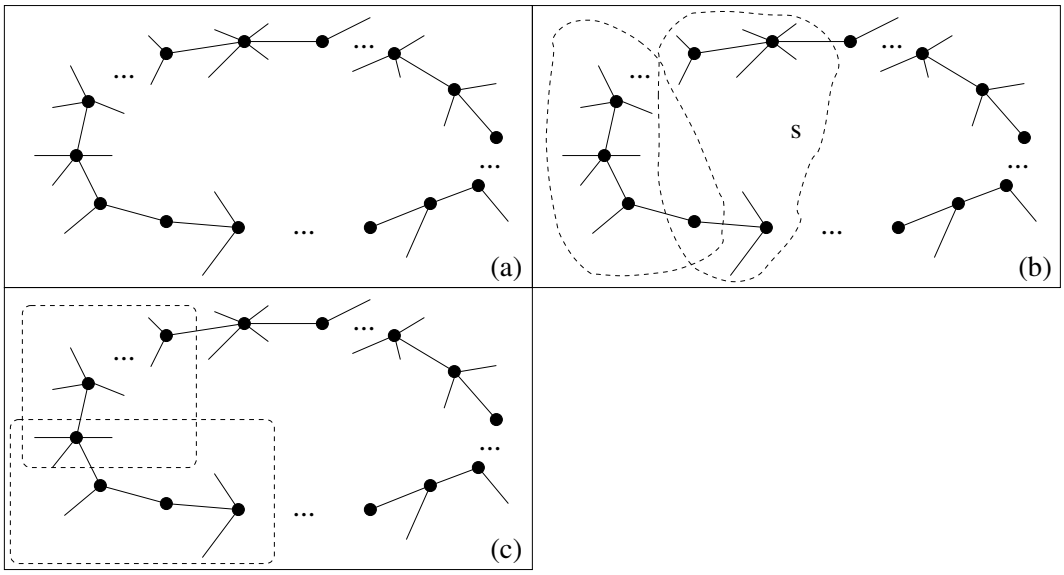

Figure 5. A possible simulated structure. The direction of each link has been omitted.

should also be sparsely connected so that an efficient runtime representation (called a *linkage tree*) [5] can be derived. This is the efficiency criterion.

One obvious method of simulation is to simulate a very large BN and then split it into subnets. Due to the large number of variables to be simulated, such an approach will have a very low probability to produce MSBNs that satisfy all criteria outlined above. Consider a simulated BN structure as illustrated in Figure 5a. It contains an undirected cycle. Note that the existence of such a cycle in a randomly generated connected DAG is very high (e.g., using the methods in Spirtes [7] or Xiang and Miller [8]). If it is split as shown in Figure 5b, the subnet S will be disconnected. If it is split as shown in Figure 5c, the hypertree condition will be violated.

Another obvious method is to first simulate a set of subnets randomly and then check whether they collectively satisfy the requirements (mostly the legal and the efficiency requirements). Although the method seems different from the above, a similar problem occurs. Both methods follow a more general approach of generate-and-test. Because the requirements were not taken into account during generation, the methods have a high likelihood of producing illegal MSBNs that have to be discarded. As a result, a very large number of simulations need to be run before a few satisfactory MSBNs can be produced.

An alternative approach is to use the requirements to constrain the simulation process. The simulation process is viewed as a sequence of decisions rather than a one-shot generation. Each decision is constrained by some requirements in the hope that the final outcome is more likely to be satisfactory. In the case that the outcome is unsatisfactory, instead of discarding the entire outcome, it is regarded as the result of some incorrect decisions. The simulation process will then "backtrack" to the corresponding decision points and resimulate the remaining.

In this work, we explore the extreme case of the alternative approach. We use the requirements to constrain the simulation process to such a degree that almost all decisions made are "correct" in the sense that the simulation makes almost no backtracking and every graphical model simulated is a satisfactory MSBN. The following top-down generation is adopted: first, the *macro* aspects of the structure are generated, and *micro* details are added stepwise. Algorithm 1 shows the top-level algorithm.

Algorithm 1

1. simulate a hypertree topology;
2. simulate hypernodes and hyperlinks as junction trees in a breadth-first fashion;
3. convert the junction tree at each hypernode into a connected DAG;
4. simulate probability parameters;

The hypertree topology is determined at the onset without specifying the details of any hypernodes and hyperlinks. Next, for each hyperlink, an "envelope" structure (called a *junction tree*) is simulated. We introduce the concept below.

Given a set X, one can define a collection Ω such that each element in Ω is a subset of X, and the union of all elements in Ω is exactly X. A junction tree T

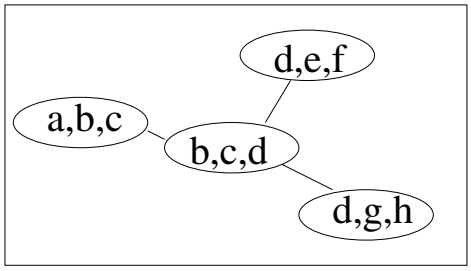

Figure 6. A junction tree of four clusters

over Ω is a graph whose nodes (called *clusters*) are labeled by elements of Ω such that there is a unique path in T between each pair of clusters, and the intersection of every pair of clusters is contained in every cluster on the path between them. The intersection of each adjacent cluster in T is called a *separator*. Figure 6 shows a junction tree, where $X = \{a, b, c, d, e, f, g, h\}$, $\Omega = \{\{a, b, c\}, \{b, c, d\}, \{d, e, f\}, \{d, g, h\}\}$. For our purpose, elements of clusters correspond to domain variables. When two variables x and y are separated by a separator S in a junction tree, it represents that x and y are conditionally independent given the value of variables in S.

The purpose of a junction tree envelope for each subnet is to define a minimum set of conditional independence relations among variables (in terms of graphical separation). This minimum set of independence relations will be augmented later to form all graphically identifiable independence relations in the subnet when the junction tree is transformed into a DAG. Therefore, the junction tree envelope controls the sparseness of the final DAG, which is an important factor for the efficiency of the resultant subnet. In addition, the junction tree will also bound the sparseness of the d-sepset (which affects the efficiency of interagent message passing). In the second step of the algorithm, a junction tree is simulated for each subnet, and a junction tree is simulated for each d-sepset (called a *linkage tree*).

In the third step, the junction tree for each subnet is converted into a DAG, and the dependence structure of an MSBN (a hypertree MSDAG) is completed. In the last step, the numerical probability distribution associated with each variable is simulated.

The subsequent sections of this article correspond to these major steps as follows: section 4 presents an algorithm to simulate the hypertree topology. Sections 5 and 6 describe algorithms for simulating the junction tree envelope for each subnet. Sections 7 through 9 deal with conversion of each junction tree into a DAG. Section 10 addresses the simulation of numerical probability distribution given the DAG dependence structure. Section 11 contains an empirical evaluation on the reachability of the simulation algorithms presented. Section 12 draws conclusions from this research.

4. Simulating Hypertree Topology

The first step in simulation is to generate the hypertree topology. The user may specify the desired total number n of hypernodes and the maximum degree d of each hypernode ($n > d \geq 2$). The algorithm is shown as follows.

Algorithm 2 (GetBoundedTree)

Input: the number $n > 0$ of nodes and the maximum degree $d (n > d \geq 2)$ of each node.
Output: an undirected tree with n nodes, each of them with a degree $\leq d$.

```
begin
    create a set V of n nodes;
    associate each node v ∈ V with a variable b_v = d;
    pick u ∈ V randomly, set V = V \ {u}, and create a
    graph G = ({u}, ∅);
    while V is not empty, do
        pick x ∈ V randomly and set V = V \ {x};
        pick a node y randomly from G where b_y > 0;
        add x to G and connect x with y;
        b_x = b_x − 1 and b_y = b_y − 1;
    return G;
end
```

In Figure 7a, V initially contains five nodes with the maximum degree $d = 3$. The first node of the graph G is randomly selected as H_2, as shown in (b). In (c), H_0 is selected and connected with H_2. Afterwards, we have $b_{H_2} = 2$ and $b_{H_0} = 2$. In (d), H_0 is chosen to be connected with the new node H_1, which results in $b_{H_0} = 1$ and $b_{H_1} = 2$. In (e), H_0 is chosen to be connected with the new node H_3, which results in $b_{H_0} = 0$ and $b_{H_3} = 2$. At this point, H_0 can no longer be chosen for connection with any new node because $b_{H_0} = 0$. Finally, H_4 is added to the tree, and the resultant G is shown in (f).

It is a simple matter to show that the algorithm will produce a tree topology with exactly n nodes, and no node will have more than d adjacent nodes.

5. Expanding a Junction Tree

Once the hypertree topology is simulated, the next step is to simulate each hypernode as a junction tree (JT) and each hyperlink as a linkage tree (LT). A linkage tree is an envelop structure for a d-sepset. Because the d-sepset is shared by the two corresponding subnets, the graphical separation relations expressed by the linkage tree must be consistent with those in the two adjacent junction trees. This means that their simulations cannot be performed independently and must be coordinated.

We therefore simulate the hypernode junction tree by traversing the hypertree in a breadth-first order and propagate the graphical separation relations as the simulation goes along. For instance, given the hypertree topology in Figure 7, the hypernodes can be generated in the order

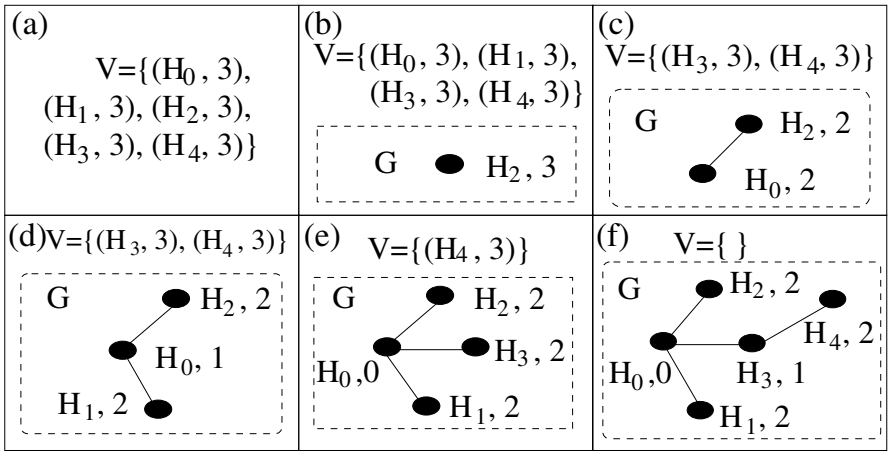

Figure 7. Illustration of GetBoundedTree

(H_0, H_2, H_3, H_1, H_4). When a hypernode H is processed, Algorithm 3 is used.

Algorithm 3

if H is the first hypernode to process, then
  initialize the JT T for H to null;
else
  initialize T to the LT between H and its adjacent, already processed hypernode;
for each adjacent hypernode H' not yet processed, do
  select a subtree S from T
  expand S and assign the resultant S' as the LT between H and H';
  merge S' into T;
expand T;

Note that the initialization (the if-else step) propagates the graphical separation from the hypernodes already processed. The body of the loop propagates the graphical separation to unprocessed hypernodes through subtree selection. The expansion operation is presented in Algorithm 4, and the merging operation is presented in the next section. These operations are developed to satisfy an important property: they respect the graphical separation relations of the input structures. The notation $|V|$ is used to denote the cardinality of a set V.

The initial *if* section of Algorithm 4 produces a single-cluster JT when the input JT T is null. The *while* loop expands T by either expanding an existing cluster or creating a new cluster, controlled by the Boolean variable b.

Note that when $b = false$, because $0 < |X| < K$, it is always possible to select S (a separator) such that $|S| > 0$ and $|S \cup X| \leq K$. On the other hand, when $b = true$, a cluster Q may not exist such that $|Q \cup X| \leq K$. In such case, the currently selected X is dropped, causing backtracking. Such backtracking has a very local effect and has a very high probability to succeed in the next few iterations (when new X and b values are selected).

Algorithm 4 (ExpandJunctionTree)

Input: a set $V \neq \emptyset$ and a JT T over a set U such that $U \cap V = \emptyset$, the maximum cluster size $K (\leq |V|)$.
Output: a JT over $U \cup V$ with separators in T preserved and with the maximum cluster size $\leq K$;
begin
  if T is null, then
    randomly choose $C \subseteq V$ $(0 < |C| \leq K)$;
    set $V = V \setminus C$ and set T to have a single cluster C;
  while V is not empty, do
    randomly choose $X \subseteq V$ $(0 < |X| < K)$ and set $V = V \setminus X$;
    randomly set the value of a boolean variable b;
    if $b = true$, then
      search for a cluster Q in T $(|Q \cup X| \leq K)$;
      if Q is found, `expand` cluster $Q = Q \cup X$;
    else
      randomly pick a cluster Q in T;
      randomly choose $S \subset Q$ such that $|S| > 0$ and $|S \cup X| \leq K$;
      `create` a new cluster $S \cup X$ and connect it with Q in T;
  return T;
end

In Figure 8a, a set V of variables used to expand a junction tree T is shown. In (b), a subset $\{g, h\}$ of V is removed to expand cluster C_1. In (c), i is removed from V to expand cluster C_2, and in (d), j is used to expand C_3. In (e), k is used to form a new cluster C_4. Note that the separator $\{b, d, i\}$ between C_2 and C_4 is randomly chosen. In (f), the remaining variables form a new cluster C_5 in T.

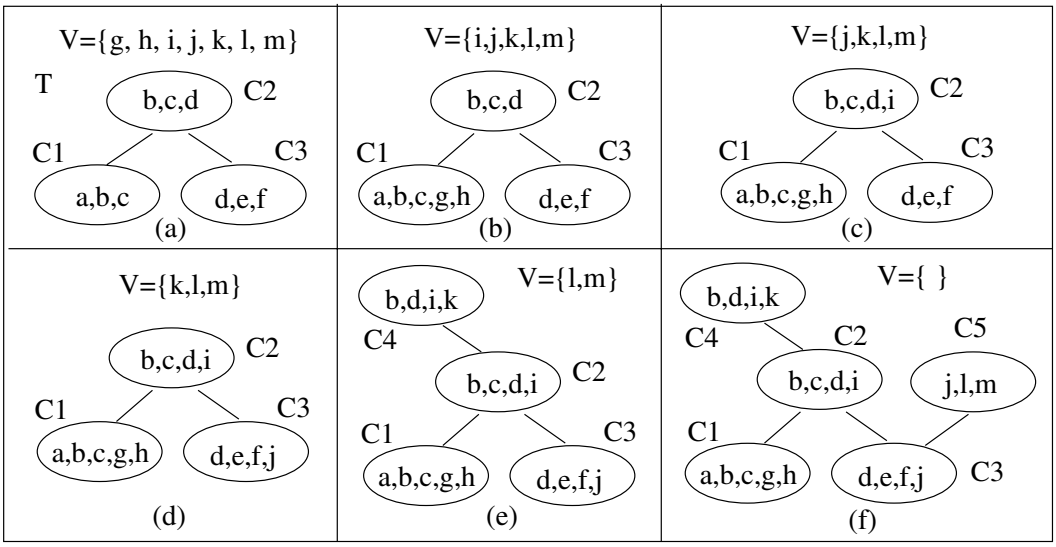

Figure 8. Illustration of ExpandJunctionTree

Proposition 5 shows that ExpandJunctionTree preserves the graph separation relations defined by the input junction tree T.

PROPOSITION 5. ExpandJunctionTree will halt and return a junction tree over the set $U \cup V$ of variables, and all separators in T are preserved.

Proof. To prove that the returned T is a junction tree, we use induction on the number of iterations of the *while* loop. Before the loop is entered, T is a junction tree that has at least one cluster. Assume that after the k'th iteration, T is a junction tree. Consider the following $k+1$'th iteration.

Let T at the beginning of the $k+1$'th iteration be over the set U'. At the $k+1$'th iteration, one of two actions could be taken: expanding an existing cluster or creating a new cluster. When an existing cluster Q is expanded, a subset X of V is unioned into Q. Since $U' \cap X = \emptyset$, the expansion does not affect any of the separators in T. Therefore, after the expansion, T is still a junction tree.

When a new cluster is created, it is formed by the union of a subset X of V and a proper subset S of an existing cluster Q. The new cluster Q' is connected to Q as a terminal cluster. Now consider any cluster Z in T other than Q and Q' such that $Z \cap Q' \neq \emptyset$. We show that $Z \cap Q'$ is contained in every cluster on the unique path between Z and Q'. Since $Z \cap X = \emptyset$, we have $Z \cap Q' = Z \cap S$. Since $S \subset Q$ and T is a junction at the start of the $k+1$'th iteration, $Z \cap S$ is contained in every cluster on the unique path between Z and Q'.

Since each cluster in the returned T consists of either elements from clusters in the original input T or elements in V, the returned T is over the set $U \cup V$.

Finally, if a variable in V is added to original clusters of T, it is added to exactly one such cluster. Hence, the separator between each pair of original clusters remains the same. □

We indicate that ExpandJunctionTree is both necessarily constrained and sufficiently flexible. It is sufficiently flexible to produce a variety of expanded JTs by both expanding existing clusters and creating new ones. It is necessarily constrained to ensure that the graph separation relations in the input JT are preserved. It would not be able to ensure this if, for example, two existing clusters are allowed to be joined into one.

Lemma 6 ensures that adjacent clusters in T are still adjacent after expansion. It is used later to analyze the merging operation, which is used in connection with ExpandJunctionTree.

LEMMA 6. Let T be a junction tree expanded from a junction tree R by ExpandJunctionTree. Then the topological adjacency between original clusters of R remains the same after R is expanded into T.

Proof. The lemma is true since no cluster is ever inserted in between existing clusters during the expansion. □

When a group of clusters has identical adjacency among themselves in two JTs, we say that the group has the identical *internal* adjacency in the JTs involved.

Proposition 7 ensures that the cluster size is bounded, and its proof is straightforward. The maximum cluster size is an important parameter for studying the inference efficiency of the simulated MSBN and hence should be specified by the user of the simulation. We emphasize that this parameter will be difficult to control and satisfy in a generate-and-test simulation.

PROPOSITION 7. Let T be a junction tree expanded from a junction tree R by ExpandJunctionTree with parameter K. If the size of the maximal cluster in R is no larger than K, then that in T is no larger than K.

6. Merging Junction Trees

In this section, we present the merging operation used in Algorithm 3. Note that in Algorithm 3, a subtree from a JT is expanded to form a (nonempty) LT, which is then merged back to the JT. The key constraint for the merging operation is to preserve the graph separation relations in both structures. We first introduce the necessary terminology.

Let T be a JT, R be a (possibly null) subtree of T, and T' be an expansion of R. We shall call T a *source* JT and T' a *partial expansion* of T. We associate a Boolean flag *marked* with each cluster in a JT. A cluster is said to be *marked* if the flag is true; otherwise, it is *unmarked*. Let Y' be a partial expansion of a JT Y. We mark the clusters in Y' as follows: if a cluster C in Y' is identical to a cluster Q in Y or is *expanded* (see the highlighted term in ExpandJunctionTree) from a cluster Q in Y, C will be marked, and Q is called the *source* cluster of C. If a cluster in Y' is *created* (see the highlighted term in ExpandJunctionTree), it will be unmarked. With clusters thus marked, we say that Y' is a *marked* partial expansion of Y. Note that there is an one-to-one mapping between marked clusters in Y' and clusters in Y. Algorithm 5 defines the merging operation.

Algorithm 5 (MergeJunctionTree)

Input: a source junction tree T, a marked partial expansion T' of T.
Output: a junction tree from merging T and T', with separators in T and T' preserved.
begin
  search T' for a marked cluster C with a single adjacent marked cluster C';
  while C is found, do
    delete the link between C and C', which split T' into two subtrees;
    denote the subtree rooted at C by R_C and denote the other subtree by T'';
    search T for a cluster Q that is the source cluster of C in R_C;
    merge R_C into T by replacing Q with C;
    search T' for a marked cluster C with a single adjacent marked cluster C';
  if C is not found, then
    denote the single marked cluster in T' by C;
    search T for a cluster Q that is the source cluster of C;
    merge T' into T by replacing Q with C;
  return T;
end

The *while* loop handles the situations where there are more than one marked cluster. By Lemma 6, the marked clusters form a connected subtree in T'. The algorithm recursively removes a *terminal* cluster C from this subtree, together with the subtree rooted at C that contains nonmarked clusters. The subtree is connected to its source cluster Q. When the "marked subtree" is reduced to a single marked cluster, the *if* section that follows the *while* loop will be entered. The subtree rooted at the last marked cluster will be merged into T similarly.

In Figure 9, an expansion T' in (b) of T in (a) is shown. The clusters C_0, C_1, and C_2 in Figure 9b are marked clusters. In Figure 9c, T' is merged onto T. The corresponding source clusters of C_0, C_1, and C_2 in T' are Q_0, Q_1, and Q_2 in T. The tree T' is first split by deleting the link between C_0 and C_1. The cluster C_3 is connected onto Q_0. The remaining tree (containing only three clusters) is split by deleting the link between C_1 and C_2. Since $Q_1 \subset C_1$, C_1 and the subtree rooted at it (containing the cluster $\{a, k, m\}$ only) are connected onto T by replacing Q_1 with C_1. The last remaining cluster C_2 is equal to Q_2 and does not have subtrees rooted at it. The process terminates.

Proposition 8 shows that when MergeJunctionTree and ExpandJunctionTree are used together for merging and expanding, as described in Algorithm 3, the result is a JT that preserves the graphical separation relations of all the JTs involved.

PROPOSITION 8. Let $(T_0, T_0'), ..., (T_n, T_n')$ be a sequence of pairs, where T_0 is a (possibly null) junction tree, T_i' $(i = 0, ..., n)$ is a marked partial expansion of T_i obtained by applying ExpandJunctionTree, and T_i $(i = 1, ..., n)$ is a graphical structure resultant from merging T_{i-1} and T_{i-1}' by applying MergeJunctionTree. Then each T_i (T_i') is a junction tree, and T_n preserves the separators in each T_i' $(i = 0, ..., n)$.

Before proving the proposition, we illustrate the generating relations among the graphical structures in Figure 10. The structures are generated following the arrows. Each downward arrow represents an expansion, and each pair of arrows going into the same node represents a merging. Under the context of Algorithm 3, T_i' $(i = 0, ..., n)$ represent the LTs simulated and T_i $(i = 0, ..., n)$ represent the successive versions of the JT at the hypernode H. The structure T_n' is the final JT at H.

Proof. We first show that the statement holds when $n = 1$. From Proposition 5, T_0' is a JT. From Lemma 6, the marked clusters of T_0' have the same internal adjacency as in T_0. We claim that the corresponding clusters in T_1 retain the same internal adjacency as well. This is certainly true because no cluster is ever inserted between the marked clusters in T_0'. Furthermore, those subtrees that contain unmarked clusters in T_0' and are rooted at marked clusters are copied to T_0. Hence, the separators of T_0 and T_0' are preserved in T_1.

Let X and Y be two clusters in the resultant T_1. If they are both from T_0 (or T_0'), then $X \cap Y$ is contained in each cluster on the path between them since both T_0 and T_0' are JTs. If one is from T_0 and the other is from T_0', then

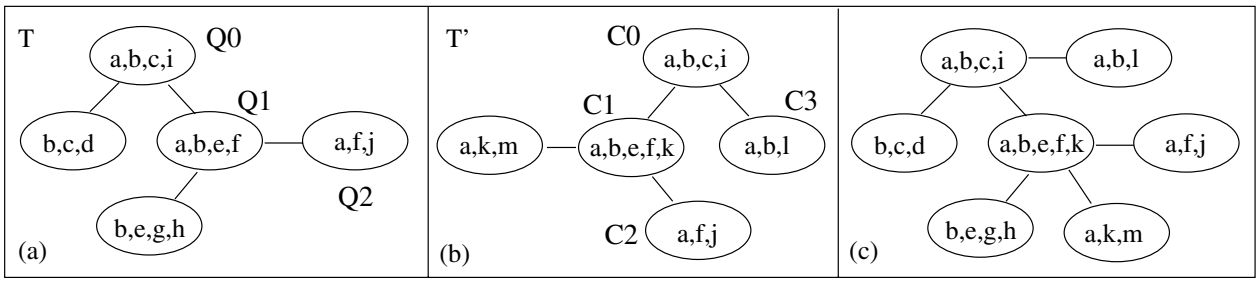

Figure 9. Illustration of MergeJunctionTree where T' contains marked clusters

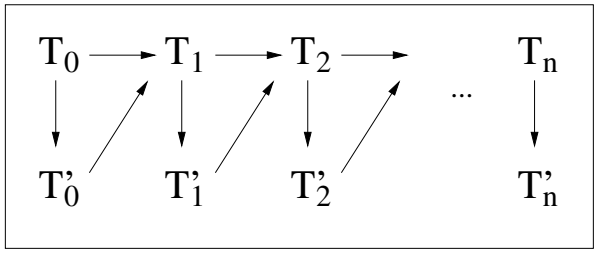

Figure 10. Illustration of relations among graphical structures

$X \cap Y \subseteq Q$. Since T_0 and T_0' are JTs and the path between X and Y has a cluster that contains Q, $X \cap Y$ is contained in each cluster on the path between them. Hence, T_1 is a JT. From Proposition 5, T_1' is also a JT.

By recursively applying the above analysis to $n > 1$, the proposition holds for any positive integer n. □

Figure 10 also illustrates how the LTs are related to the JT of a hypernode. That is, T_n' is the JT of a hypernode H, and each T_i' ($i < n$) is an LT between H and an adjacent hypernode H_i. Note that the variables contained in T_i' are a proper subset of those contained in T_n'. If H_i has another adjacent hypernode Y_i, then its JT will contain the variables in T_i' as a proper subset. Some of these variables in T_i' may also be included in the LT between Y_i and its adjacent hypernodes (other than T_n'). All other variables in Y_i will be newly generated (never been used in the previously simulated hypernodes). It is easy to see, from this simulation method, that variables shared by any two hypernodes will be contained in each hypernode along the hyperpath between them. So far, we have referred to the structure created by GetBoundedTree as *hypertree* without giving a justification. This analysis shows that the structure simulated so far indeed satisfies Definition 1 on hypertree.

We have now presented the details for the first two steps in Algorithm 1. The resultant graphical model is called a *linked junction forest* (LJF). A trivial, simulated LJF is shown in Figure 11. It consists of three JTs in hypernodes H_0, H_1, and H_2, respectively, and linkage trees L_1 and L_2.

Theorem 9 shows that an LJF is isomorphic to a hypertree, as defined in Definition 1, with each hypernode containing a junction tree and each hyperlink containing a linkage tree (which is a junction tree), and the separators in each linkage tree are preserved by the corresponding two junction trees.

THEOREM 9. Let F be an LJF simulated by the first two steps of Algorithm 1 and Algorithms 2 through 5. Let the hypernodes of F be denoted by H_i ($i = 0, 1, ...$), the set of variables contained in H_i be denoted by V_i, and the structure at H_i be denoted by T_i. Then F satisfies the following:

1. For any two hypernodes H_i and H_j, the intersection $V_i \cap V_j$ is contained in each hypernode on the path between H_i and H_j in the tree organization.
2. Each T_i is a junction tree.
3. Each hyperlink is associated with a junction tree (the linkage tree), and its separators are preserved by the junction trees in the two corresponding hypernodes.

Proof. We refer to each V_i as a subdomain. Statement 1 is true due to the breadth-first subdomain simulation by Algorithm 3. If $V_i \cap V_j \neq \emptyset$ and H_i is processed before H_j, then $V_i \cap V_j$ are the variables that H_j obtained from H_i through a series of subtree expansions, one for each hypernode between H_i and H_j. Hence, $V_i \cap V_j$ is contained in each hypernode between H_i and H_j.

Statement 2 is true due to Proposition 8. Statement 3 is true because of Propositions 5 and 8. □

In the following sections, we convert an LJF into a hypertree MSDAG while preserving the key graphical separation relations as established in Theorem 9.

7. Simulating MSDAG from LJF

Next, we consider the conversion of an LJF into a hypertree MSDAG. So far, we have simulated a set of junction trees $J_0, ..., J_{n-1}$ organized as a hypertree structure and a set of corresponding linkage trees. To convert the LJF into a hypertree MSDAG, each junction tree J_i ($i = 0, ..., n -$

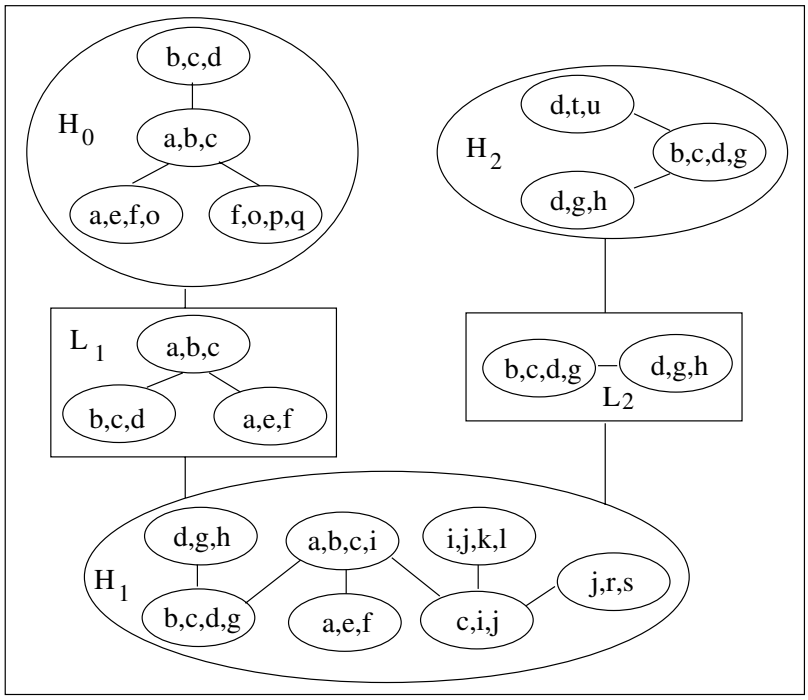

Figure 11. A trivial, simulated linked junction forest

1) needs to be converted into a local DAG D_i satisfying the following constraints. The first three constraints are obvious from Definition 3. We explain the last constraint below.

1. Each local DAG D_i is connected.
2. The union of all local DAGs is acyclic.
3. Nodes shared by adjacent local DAGs on the hypertree satisfy the d-sepset condition.
4. For each JT J_i and each local DAG D_i, if J_i is converted into an undirected graph UG_i and D_i is converted into its moral graph MG_i, then MG_i is a spanning subgraph of UG_i.

A JT J_i over its subdomain V_i can be converted uniquely into an undirected graph UG_i by the following procedure: initialize UG_i to have one node for each element $x \in V_i$ with no links. Connect each pair of nodes x and y in UG_i if x and y are contained in a cluster in J_i. An example is given in Figure 12.

A DAG D_i can be converted uniquely into an undirected graph MG_i (called a *moral graph*) by the following procedure: connect the parent nodes pairwise for each node in D_i and drop the directions of D_i. An example is given in Figure 13. Note that Figure 13b is a spanning subgraph of Figure 12b. Constraint 4 essentially requires that the local DAG D_i preserves all graphical separation relations of J_i.

We now consider how to convert each JT into a local DAG such that constraints 1 through 4 are satisfied. A directed graph is a DAG if and only if its nodes satisfy a topological order. Hence, to convert a JT into a local DAG, a topological order over elements in the subdomain needs to be defined. However, such topological orders local to subdomains do not guarantee that the union of local DAGs is acyclic (constraint 2). To ensure the global acyclicity, a topological order over the entire domain $V = \bigcup_i V_i$ is needed. We propose Algorithm 6 for the conversion.

Algorithm 6

1. Convert each JT J_i into its undirected graph UG_i.
2. Compute the graph union $UG = \cup_i UG_i$.
3. Construct a spanning tree ST of UG.
4. Convert ST to a directed graph DT by orienting its links.
5. Obtain a topological order of nodes in DT.
6. Section DT into local graphs D_i ($i = 0, ..., n-1$) according to subdomains.
7. Augment each D_i into a general (multiply connected) and connected local DAG.

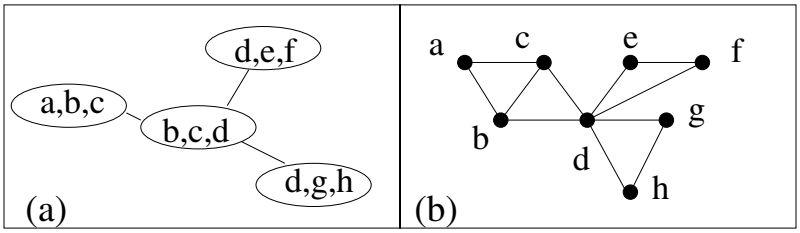

Figure 12. Converting a junction tree (JT) (a) to an undirected graph (b)

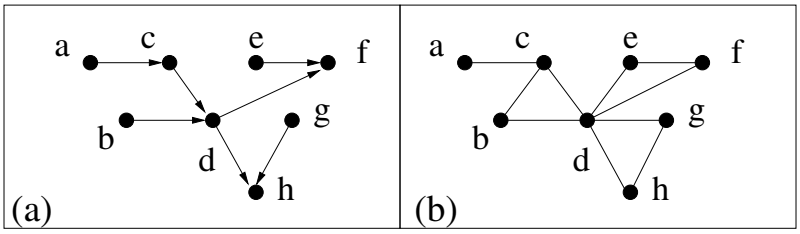

Figure 13. Converting a directed acyclic graph (DAG) (a) to its moral graph (b)

Steps 1 through 5 produce a global topological order. This order ensures that local graphs produced in step 6 are DAGs and satisfy constraint 2. This order will be respected in step 7. Hence, the final local DAGs will satisfy constraints 1 and 2.

The spanning tree at step 3 ensures that the local graphs produced in step 6 satisfy constraint 4. The execution of step 7 (presented later) also maintains this property.

Implementation of steps 1 and 2 is straightforward. A simple way to compute a spanning tree of a connected graph G is the following: copy G into a graph G' and delete all links in G'. Mark any node x in G. Find in G a marked node v with an adjacent, unmarked node y; connect v and y in G'; and mark y in G. Repeat the last step until all nodes in G are marked. The resultant G' is a spanning tree of G. Figure 14a shows the graph union UG, computed in the first two steps of Algorithm 6 from the LJF in Figure 11. The resultant spanning tree from step 3 is shown in Figure 14b.

Details for the remaining steps of Algorithm 6 are presented in subsequent sections.

8. Directing the Spanning Tree of the Union Graph

In this section, we consider step 4 of Algorithm 6. Although adding directions to a tree can, in general, be performed by arbitrarily orienting each link, doing so will violate constraint 4 described in section 7. For instance, consider the undirected graph in Figure 12b, which we duplicate in Figure 15a. Suppose we obtain the spanning tree in Figure 15b and direct it arbitrarily as shown. If we compute the moral graph of (b), as shown in (c), it violates constraint 4: the graph in (c) is not a subgraph of (a). Conceptually, in the JT of Figure 12a, elements a and d are graphically separated by the separator $\{b, c\}$. However, in Figure 15c, a and d are no longer separated by $\{b, c\}$.

Direct links in a graph can be viewed as specifying the parent set $\pi(x)$ for each node x. To satisfy constraint 4, we direct the spanning tree ST subject to the following requirement: for each node x in ST, $\pi(x)$ is chosen such that there exists a cluster C in a JT of the LJF and $\{x\} \cup \pi(x) \subseteq C$. We refer to $\{x\} \cup \pi(x)$ as the *family* of x, denoted by $fmly(x)$.

Algorithm DirectTree performs this operation recursively. Each node in ST is treated as an object, which executes the algorithm. An arbitrary node is selected (by the simulator system) to start the algorithm. In the algorithm, a *caller* is either an adjacent node of the node being called or the system.

Algorithm 7 (DirectTree)

When the caller calls in a node y to DirectTree, y does the following:

```
if caller is the system,
    y calls DirectTree in each adjacent node in ST and
    returns;
denote the caller by x and the current parent nodes of x
    by π(x);
if there exists a cluster C in the LJF such that π(x)∪
    {y} ⊆ C,
    direct the link between x and y randomly;
else direct the link from x to y;
y calls DirectTree in each adjacent node in ST except
the caller;
```

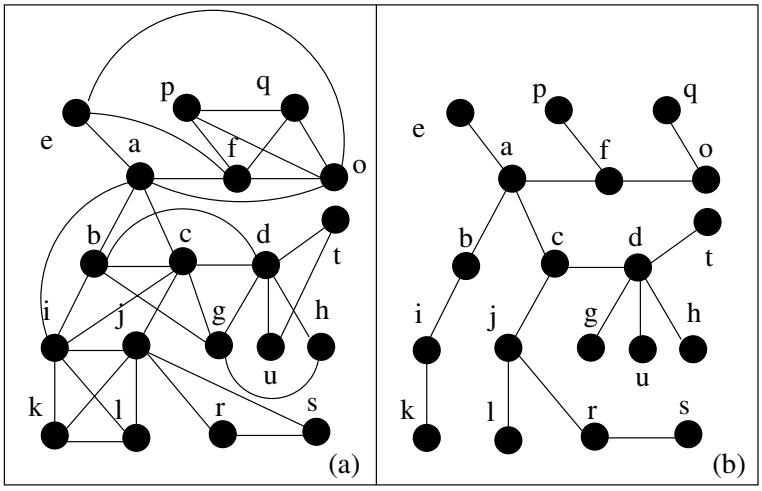

Figure 14. (a) A connected undirected graph obtained from the linked junction forest (LJF) in Figure 11. (b) A spanning tree of the graph in (a).

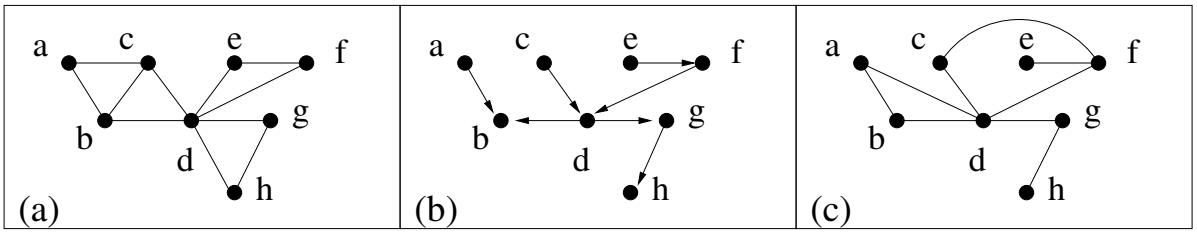

Figure 15. (a) A directed spanning tree of Figure 12b. (b) The moral graph of the graph in (a).

The first *if* statement, executed only once, handles the activation of the operation. Subsequent calls of DirectTree orient the link between the caller and the node being called. The second *if-else* statement orients the link to the maximum degree of freedom subject to the requirement dictated by constraint 4.

DirectTree is illustrated in Figure 16, where the spanning tree in Figure 14b is directed. To start, the system calls DirectTree on, say, node a. The node a simply calls nodes b, c, e, f to DirectTree. Because a has no parents at this stage, links between a and these nodes can be oriented in either direction. Assume that they are directed, as shown in (a). Suppose c subsequently calls DirectTree in d. The current parent of c is a. Because there is no cluster in Figure 11 that contains $\{a, d\}$, the link $\{c, d\}$ cannot be directed arbitrarily. It must be directed as (c, d), as shown in (a). The final result of the algorithm is shown in (b).

Proposition 10 shows that DirectTree enforces constraint 4.

PROPOSITION 10. Let UG be the undirected graph union obtained from an LJF according to Algorithm 6 (steps 1 and 2). Let ST be a spanning tree of UG. Let DT be the graph resultant by applying DirectTree to ST. Then the moral graph of DT is a spanning subgraph of UG.

Proof. It suffices to show that all links in the moral graph MG are contained in UG. The links in MG consist of those from ST and those added when DT is converted into MG (denote this set of links by L). Since ST is a spanning tree of UG, all links from ST are contained in UG.

Consider next the links in L. Any link in L is added because a node y has more than one parent. All parents of y except the first are determined through the second *if* statement of DirectTree. Because of the condition $\pi(x) \cup \{y\} \subseteq C$, the corresponding links in L are also contained in UG. □

Once the spanning tree is directed (step 4 of Algorithm 6), a topological order (step 5) of nodes can be obtained by first ordering all root nodes and then ordering remaining nodes whose parents have been ordered. In Figure 17, the directed tree in Figure 16b is shown with a topological order. The order of each node is shown in parentheses.

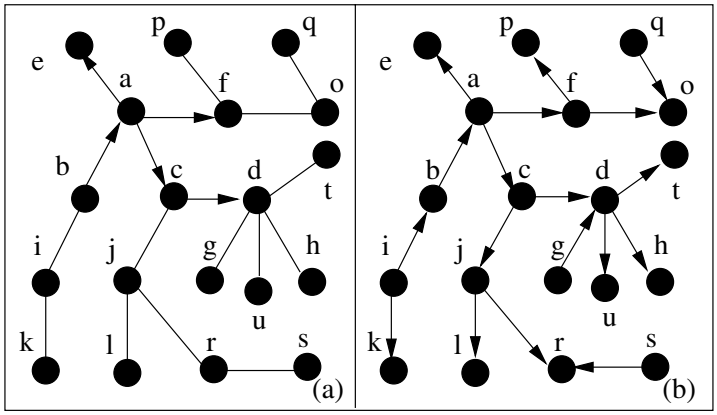

Figure 16. Illustration of DirectTree

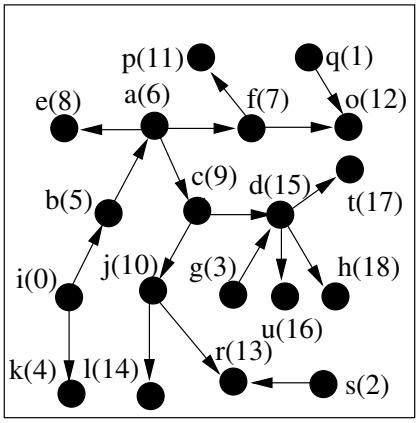

Figure 17. Topologically ordering nodes in a tree

9. Augmenting Local DAGs

Next, we consider the last two steps of Algorithm 6: sectioning the global directed tree into local DAGs according to subdomains and augmenting each local DAG. The sectioning operation is straightforward. However, a local DAG obtained by sectioning may be disconnected, violating constraint 1 in section 7. This issue is illustrated in Figure 18.

A trivial LJF is shown in Figure 18a. From the LJF, an undirected graph union can be derived and a spanning tree of the graph union can be obtained. After directing links of the tree and sorting nodes topologically, the result is shown in (b). Sectioning the tree by subdomains gives the four local graphs in (c), where the local graph for hypernode H_3 is disconnected.

To connect the local graph, a link needs to be added between b and c or d and c. If we orient the link from c toward b or d, it is against the topological order and could violate the acyclicity requirement (although this danger does not materialize in this particular example). On the other hand, if the link is directed from b or d to c, node c will have parents e as well as one of b or d. This violates constraint 3 in section 7. That is, the interface $\{c\}$ between H_2 and H_3 is not a d-sepset.

This example illustrates the possibility of disconnected local graphs and the additional difficulty to connect them while observing the topological order. Our solution to the problem is to introduce an additional node, local to the disconnected graph, as illustrated in Figure 19. In the local graph for H_3, the node v is introduced. It is connected as the child node of c and d and is given a topological order of 6 (the highest order globally). This solution renders the local graph connected and respects the topological order as well as the d-sepset constraint.

In summary, after sectioning of a global directed tree, the local DAG obtained may be disconnected. Therefore, augmenting each local DAG involves (1) making the local DAG connected and (2) making the local DAG multiply connected in general. The recursive algorithm AugmentDAG performs this task. Each hypernode on the LJF is treated as an object, which executes the algorithm. An arbitrary hypernode on the LJF is selected (by the simulator system) to start the algorithm. The hypernode then activates adjacent hypernodes to execute the algorithm, and the processing propagates through the hypertree. In the algorithm, a *caller* is either an adjacent hypernode H' with its local graph G' or the simulator system. The function $ord(x)$ returns the topological order of a node x. If $ord(x) < ord(y)$, then y cannot be the parent of x in a DAG.

Algorithm 8 (AugmentDAG)

When caller calls AugmentDAG in a hypernode H with local graph G, it does the following:

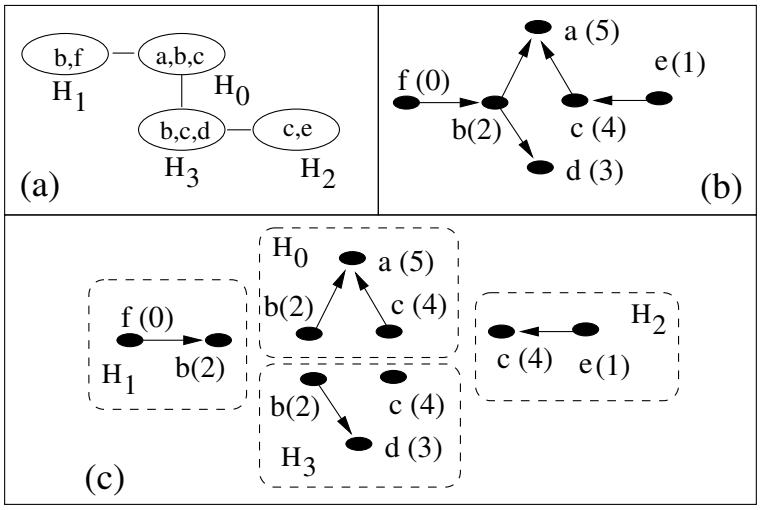

Figure 18. (a) A trivial linked junction forest. (b) A directed spanning tree derived from (a). (c) Local graphs obtained by sectioning (b).

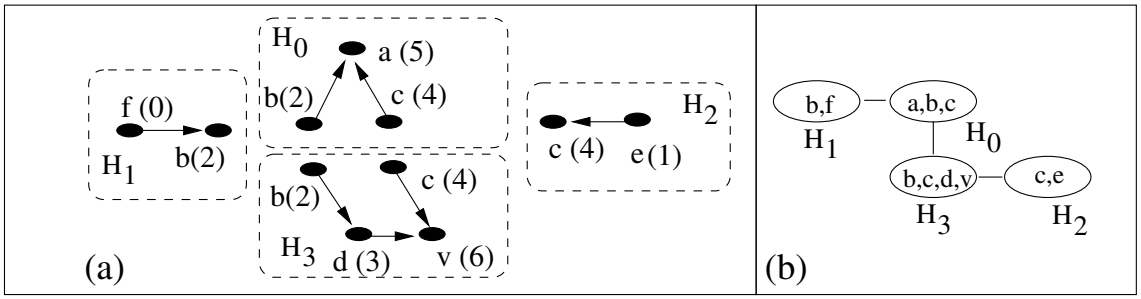

Figure 19. (a) A child node v is introduced to make the local graph in H_3 connected. (b) The corresponding hypertree.

if caller is the system, let G' denote a null graph;
else denote the local graph in caller by G';
denote the nodes shared between G and G' by S;
copy all arcs in G' between nodes in S to G;
if G is disconnected, connect its components;

for every pair of nodes u, v in G such that the pair is not contained in G', do
  if u and v are not directly connected
    if there exists a cluster C in the LJF such that $\{v\} \cup fmly(u) \subseteq C$
      if ord$(v) < ord(u)$, decide randomly whether the arc (v, u) will be added;
    if there exists a cluster C in the LJF such that $\{u\} \cup fmly(v) \subseteq C$
      if ord$(u) < ord(v)$, decide randomly whether the arc (u, v) will be added;
H calls AugmentDAG in each adjacent hypernode except the caller;
When a hypernode H is called to AugmentDAG, first it makes arcs between shared nodes S in its local graph G consistent with those in G' (by the copy operation). It then makes sure that G is connected.

Afterwards, each pair of nodes (not directed connected) is examined and connected randomly subject to two constraints. The first constraint is $\{u\} \cup fmly(v) \subseteq C$ or $\{v\} \cup fmly(u) \subseteq C$, which is implied by constraint 4 in section 7. The second constraint is to respect the topological order, which is implied by constraint 2 in section 7.

The pseudocode of AugmentDAG did not detail the processing when G is disconnected. Essentially, it is handled using the idea outlined earlier in this section: given a pair of (disconnected) components of G, a pair of nodes, one from each component, will be searched subject to the two constraints described above. If the pair is found, they will be connected respecting the topological order. Otherwise, a pair of nodes, contained in a cluster in the LJF, will be selected randomly, and a common new child node will be introduced.

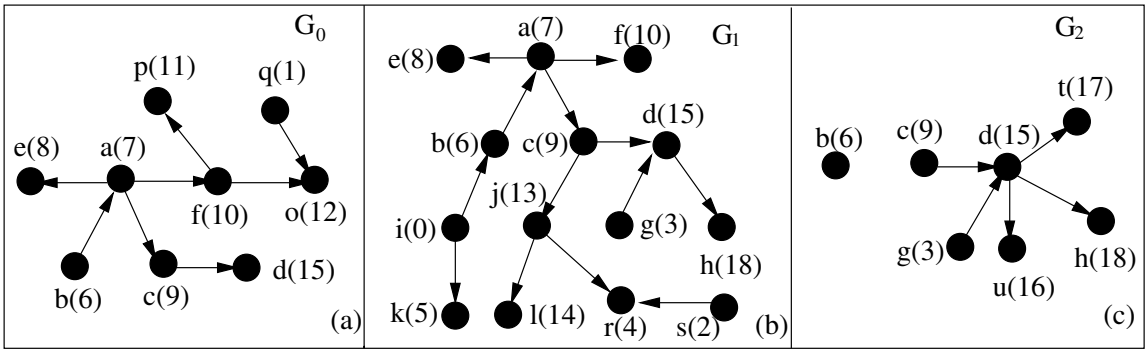

Figure 20. Local directed acyclic graphs (DAGs) obtained by sectioning the spanning tree in Figure 17. G_i ($i = 0, 1, 2$) corresponds to the local graph of H_i in Figure 11.

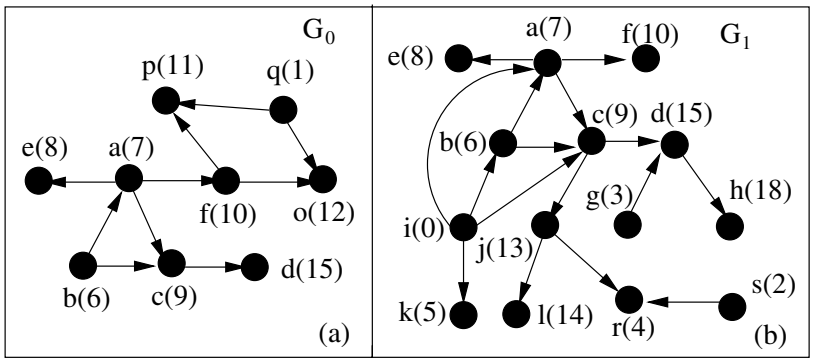

Figure 21. Local directed acyclic graphs (DAGs) G_0 and G_1 are augmented

As an example, consider the spanning tree in Figure 17. It is sectioned into three local DAGs, as shown in Figure 20, corresponding to subdomains defined in Figure 11. Note that the local DAG G_2 is disconnected.

In Figure 21, local DAGs G_0 and G_1 are augmented. In (a), arcs (b, c) and (q, p) are added. In (b), arc (b, c) is copied from (a), and two new arcs, (i, a) and (i, c), are added. Figure 22 shows the augmentation of G_2. Its disconnected components are connected in (a) by an arc copied from G_1 in Figure 21b. An arc (t, u) is then added in (b).

Theorem 11 shows that a legal MSBN is simulated by the suite of algorithms that we have presented so far.

THEOREM 11. Let F be an LJF simulated by the first two steps of Algorithm 1 and Algorithms 2 through 5. Let M be the structure simulated from F according to Algorithms 6 through 8.

Then M is a hypertree MSDAG and preserves the graphical separation relations among variables specified by F.

Proof. Given Theorem 9, it suffices to show that constraints 1 through 4 specified in section 7 are satisfied.

Constraint 1 is satisfied if each local graph is a DAG and is connected. Each local graph is a DAG because it is simulated following a topological order. It is connected due to the connection step in AugmentDAG.

Constraint 2 on global acyclicity is satisfied because (1) each local DAG is simulated by following a global topological order, and (2) each newly introduced node is given the highest global order and thus extends the order consistently.

Constraint 3 on d-sepset is also satisfied because of statement 3 in Theorem 9 and the **[in?]** respect of clusters in F by the simulation of the spanning tree and by AugmentDAG. The d-sepset condition can only be violated if a node x shared by multiple hypernodes has two parents y and z, but no hypernode contains both y and z. This will never occur because DirectTree and AugmentDAG together require that x, y, and z be contained in a single cluster of F, and statement 3 in Theorem 9 implies that this cluster is a cluster in a linkage tree.

Finally, constraint 4 is satisfied due to Proposition 10 and the **[in?]** respect of clusters in F by AugmentDAG. □

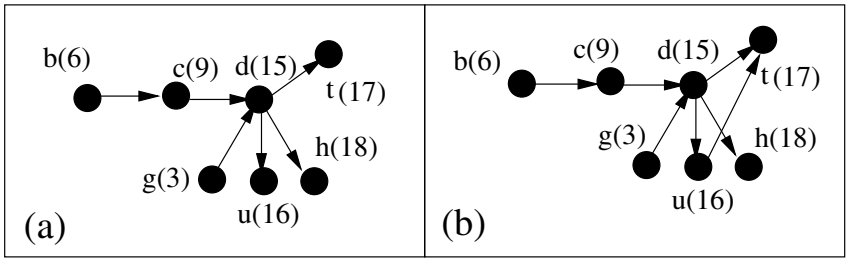

Figure 22. Local directed acyclic graph (DAG) G_2 is augmented

10. Simulating Conditional Probability Distributions

Once a hypertree MSDAG is simulated, to complete the simulation of an MSBN, we need only to simulate the conditional probability distributions. According to Definition 4, this amounts to simulation of $P(v|\pi(v))$ for each variable v that is unique in a subdomain and $P(x|\pi(x))$ for each shared variable x, where $\pi(x)$ denotes all parent variables of x in the entire universe. The distributions, however, cannot be arbitrarily simulated. Consider a variable x with parent variables y and z. If the simulated distribution $P(x|\pi(x))$ satisfies $P(x|y, z) = P(x|y)$, then the arc (z, x) can be deleted, and the resultant simplified MSDAG is still a correct dependence structure for the simulated jpd. That is, the original MSDAG is not a *minimal I-map* [4] of the simulated jpd. To ensure that the simulated MSDAG represents a minimal I-map of the simulated jpd, we test each simulated $P(v|\pi(v))$ before accepting it. The test can be described as follows.

Given x and $\pi(x)$, let $s(x)$ be a proper subset of $\pi(x)$ and $n(x)$ be the set difference $\pi(x) \setminus s(x)$. If for each configuration of $\{x\} \cup s(x)$, the distribution $P(x|s(x), n(x))$ has identical values over all configurations of $n(x)$, then the simulated $P(x|\pi(x))$ will be rejected and resimulated.

It appears that for each $\pi(x)$, we need to perform the test for every subset of $\pi(x)$. However, this amounts to an exponential number of tests in the cardinality of $\pi(x)$. The following proposition provides a condition by which a linear number of tests in the cardinality of $\pi(x)$ will be sufficient. The proposition refers to a property of probability called weak union [4], where $n(x)$, y, and $w(x)$ form a partition of $\pi(x)$: if $P(x|n(x), y, w(x)) = P(x|n(x))$, then $P(x|n(x), y, w(x)) = P(x|n(x), w(x))$.

PROPOSITION 12. Let $P(x|\pi(x))$ be a conditional probability distribution. If every subset $s(x) \subset \pi(x)$ ($|s(x)| = |\pi(x)| - 1$) satisfies $P(x|s(x)) \neq P(x|\pi(x))$, then every subset $n(x) \subset \pi(x)$ ($|n(x)| \leq |\pi(x)| - 2$) also satisfies $P(x|n(x)) \neq P(x|\pi(x))$.

Proof. We prove by contradiction. Assume that there exists an $n(x) \subset \pi(x)$ ($|n(x)| \leq |\pi(x)| - 2$) such that $P(x|n(x)) = P(x|\pi(x))$. Denote $\pi(x) = n(x) \cup \{y\} \cup w(x)$, where $n(x)$, $\{y\}$, $w(x)$ are disjoint. Since $|n(x)| \leq |\pi(x)| - 2$, $\{y\}$ and $w(x)$ do exist.

By assumption and weak union, we have $P(x|\pi(x)) = P(x|n(x), w(x))$. Since $|n(x) \cup w(x)| = |\pi(x)| - 1$, we have $P(x|s(x)) = P(x|\pi(x))$, where $s(x) = n(x) \cup w(x)$ and $|s(x)| = |\pi(x)| - 1$: a contradiction to the condition $P(x|s(x)) \neq P(x|\pi(x))$ for each $s(x) \subset \pi(x)$. □

Proposition 12 suggests that it is sufficient to perform the test described earlier in this section for each subset $s(x) \subset \pi(x)$ ($|s(x)| = |\pi(x)| - 1$). Thus, only up to $|\pi(x)|$ tests are needed.

With the conditional probability distributions simulated, a complete MSBN is generated.

11. On the Reachability of the Simulation

As discussed in section 3, proving that an MSBN simulator system can generate every legal MSBN with nonzero (or equal) probability is quite difficult. We have instead evaluated the reachability of the simulator system through an empirical study.

The set of algorithms presented is implemented in Java. We present one simulated MSBN as an example result and then the statistical data from one set of 100 simulations. Figures 23 through 25 show the result of one simulation. The simulation was run with the following parameters:

Number of subnets:	5
Maximum number of adjacent agents:	3
Maximum cluster cardinality:	12
Maximum initial first subdomain cardinality:	20
Maximum number of variables per expansion:	20

The total universe contains 141 variables. The subdomains have the cardinalities 57, 48, 58, 53, and 58, respectively. Individual variables have between 0 to 7 parent variables and between 0 to 38 child variables. The four d-sepsets contain 13, 15, 23, and 25 variables, respectively. For instance, the d-sepset between subnets G_0 and G_2 is

$$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{14}, v_{15}\}.$$

Next, we present the statistical data from one set of 100 simulations. The simulation was run with the follow-

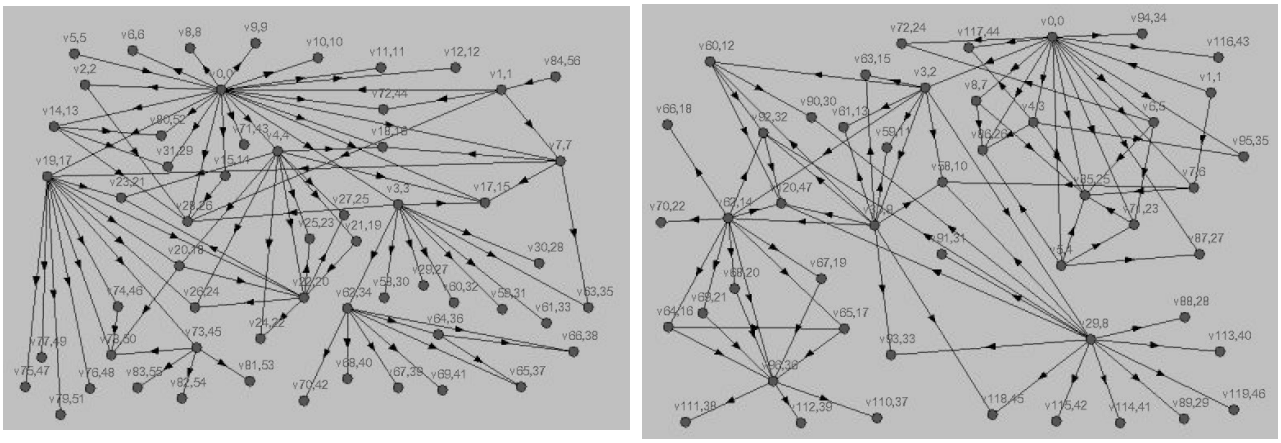

Figure 23. Subnet G_0 (left) and subnet G_1 (right)

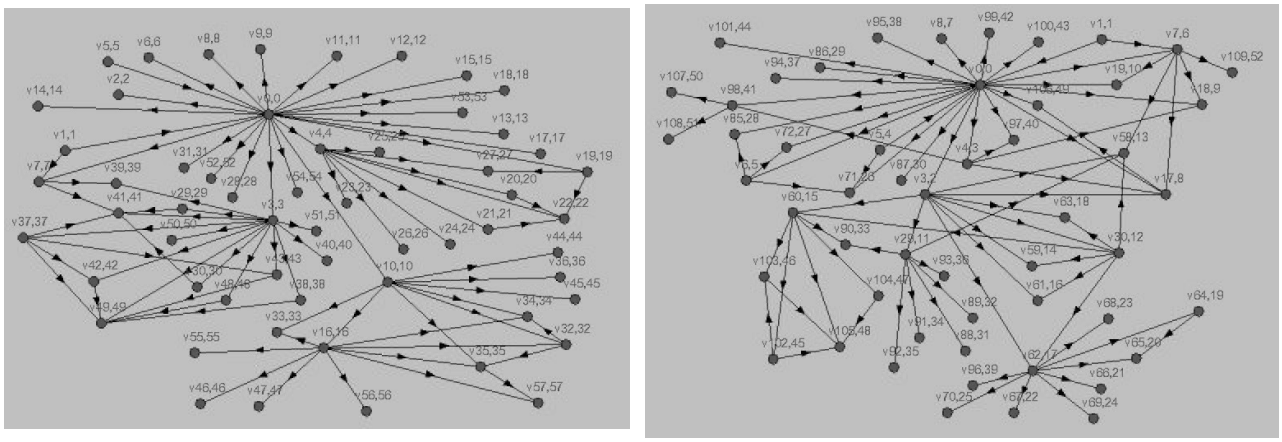

Figure 24. Subnet G_2 (left) and subnet G_3 (right)

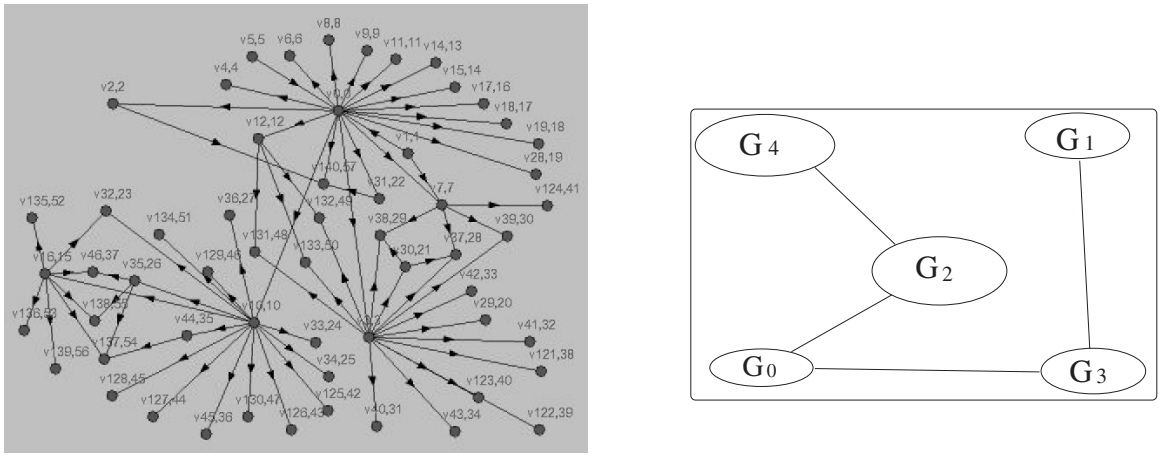

Figure 25. Subnet G_4 (left) and hypertree organization of the multiply-sectioned Bayesian network (MSBN) (right)

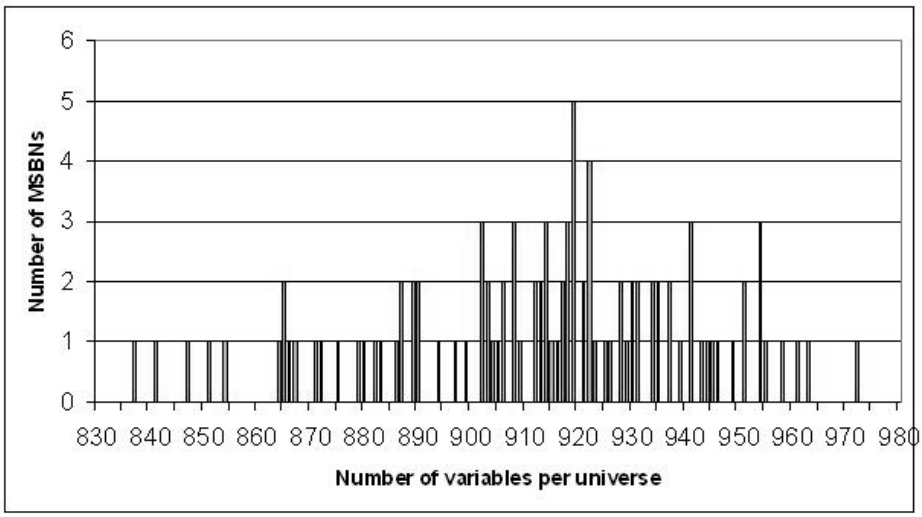

Figure 26. Total number of variables in each universe

ing parameters (similar results were obtained with other parameters):

Number of subnets:	20
Maximum number of adjacent agents:	5
Maximum cluster cardinality:	12
Maximum initial first subdomain cardinality:	20
Maximum number of variables per expansion:	30

Figure 26 shows the distribution of the total number of variables in the universe (the set of domain variables for an entire MSBN). The number of variables ranges from 837 per universe to 972, with a somewhat bell-shaped distribution peaking at 919 variables (with five MSBNs simulated).

Figure 27 shows the distribution of the number of variables per subnet. The number of variables per subnet ranges between 35 and 235, demonstrating a wide range of subdomain cardinalities. A large number of subnets (about 65% of the 2000 subnets) have a subdomain with a size between 60 and 130 variables.

Figures 28 and 29 examine the adjacency of individual variables. Figure 28 shows the number of parent variables. About 50% of the variables have a single parent variable, and about 20% have two parents. As we have specified the maximum cluster cardinality to be 12, this sets an upper bound for the number of parents each variable can have to 11 variables.

Figure 29 shows the distribution of the number of child variables. A wide range of numbers of child variables was simulated. As the number of child variables increases from 0 to 192, the corresponding number of variables decreases rapidly. To illustrate the magnitude of the change, three subfigures are used. The upper left subfigure shows the distribution when the number of child variables increases from 0 to 6. The upper right subfigure shows the distribution when the number of child variables increases from 6 to 22, and the bottom subfigure shows that from 22 to 192.

Figure 30 examines the interface between agents/subnets. On the left, the distribution of the number of adjacent subnets per subnet is shown. We have specified the maximum number of adjacent subnets to be 5, and the figure shows how the size of adjacency is distributed in the simulation. On the right, the distribution of the d-sepset size is presented. The d-sepsets of sizes between about 10 and 45 have the highest concentration. A single spike appears at 12 variables. This corresponds to our specification for the maximum cluster size. It represents the number of single-cluster linkage trees where the cluster is at the upper bound of its cardinality.

These experimental data clearly demonstrate that the simulation algorithms are capable of producing legal MSBNs with a wide range of topologies. It is a simple matter to show that the set of algorithms collectively provides an *efficient* simulator system. The simulation of the hypertree MSDAG is of polynomial time complexity. The complexity to simulate the probability distributions is linear on the cardinality of the universe and is exponential on the maximum cluster size.

12. Conclusion

Simulation provides an efficient alternative to knowledge engineering and can fuel experimental studies of algorithms for multiagent probabilistic reasoning with a variety of graphical models.

We have presented a suite of algorithms that can randomly simulate MSBNs with a minimum amount of backtracking. Simulation of an MSBN requires the generation

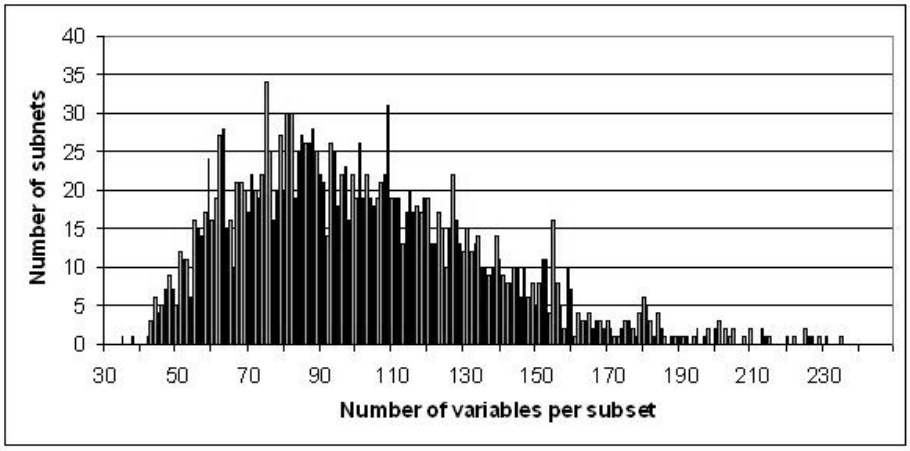

Figure 27. Number of variables per subnet

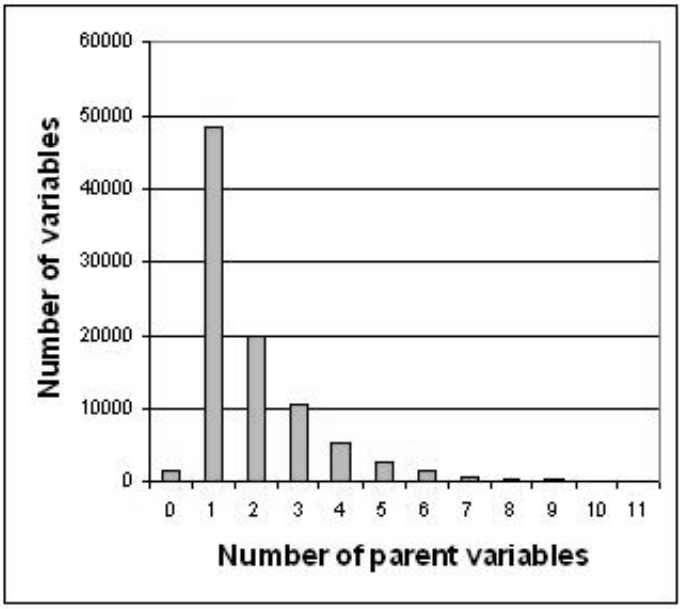

Figure 28. Number of parents per variable

of a set of interrelated graphical structures over a very large set of nodes/variables and a joint probability distribution defined over them. A set of constraints needs to be satisfied. This causes a pure generate-and-test approach to fail and to regenerate frequently. To reduce such backtracking, we proposed a top-down and stepwise-constrained approach for simulation. At each step of simulation, decisions are made subject to the necessary constraints while retaining a sufficient degree of freedom. The simulation process is proven to always succeed while the experimental study demonstrates that legal MSBNs of a wide variety of topological structures can be simulated. The suite of algorithms also provides a rich set of new graphical manipulation operations.

13. Acknowledgments

This work is supported by research grants to the first and the third authors from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

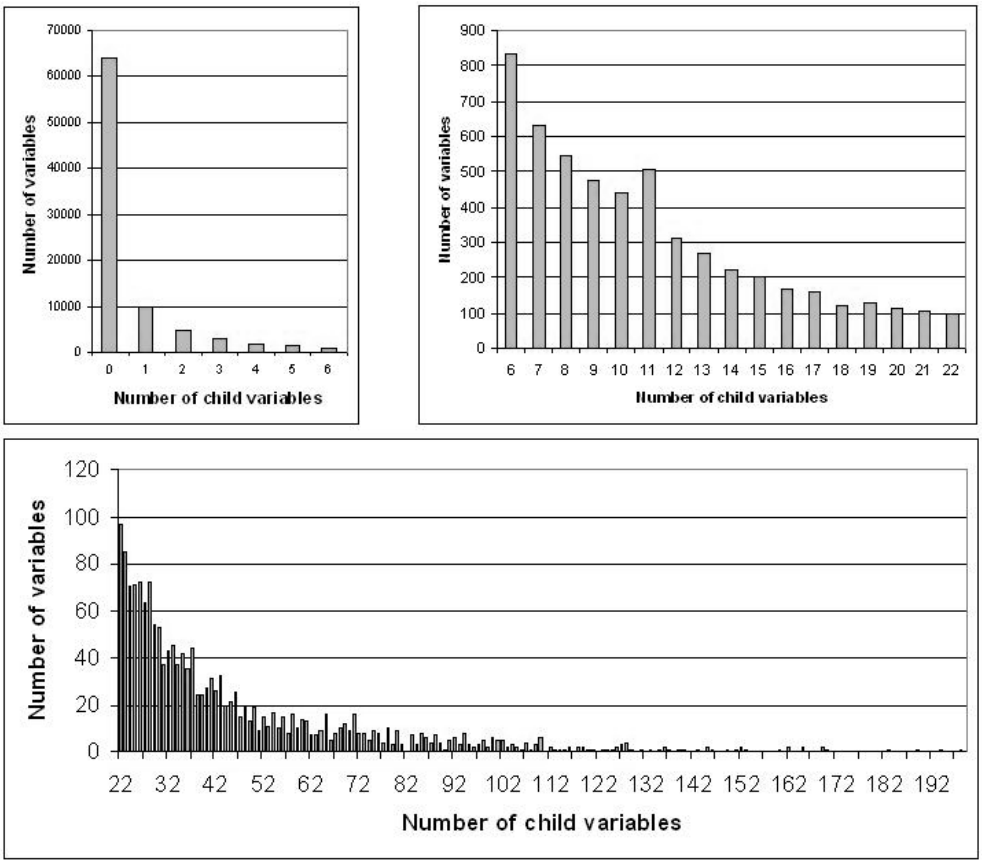

Figure 29. Number of children per variable

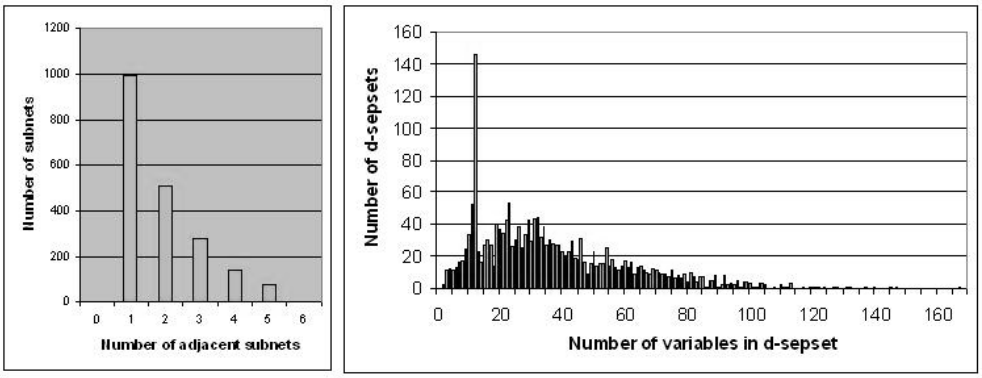

Figure 30. Left: Number of adjacent subnets per subnet. Right: Number of variables per d-sepset.

14. References

[1] Sycara, K. P. 1998. Multiagent systems. *AI Magazine* 19(2): 79-92.

[2] Wooldridge, M., and N. R. Jennings. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10 (2): 115-52.

[3] Lesser, V. R., and L. D. Erman. 1980. Distributed interpretation: A model and experiment. *IEEE Transactions on Computers* C-29 (12): 1144-63.

[4] Pearl, J. 1988. *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. New York: Morgan Kaufmann.

[5] Xiang, Y. 1996. A probabilistic framework for cooperative multiagent distributed interpretation and optimization of communication. *Artificial Intelligence* 87 (1-2): 295-342.

[6] Xiang, Y., and H. Geng. 1999. Distributed monitoring and diagnosis with multiply sectioned Bayesian networks. In *Proceedings of the AAAI Spring Symposium on AI in Equipment Service, Maintenance and Support*, Stanford, CA, pp. 18-25.

[7] Spirtes, P., C. Glymour, and R. Scheines. 1993. *Causation, prediction, and search*. Berlin: Springer-Verlag.

[8] Xiang, Y., and T. Miller. 1999. A well-behaved algorithm for simulating dependence structures of Bayesian networks. *International Journal of Applied Mathematics* 1 (8): 923-32.

[9] Breese, J. S. 1992. Construction of belief and decision networks. *Computational Intelligence* 8 (4): 624-47.

[10] Goldman, R. P., and E. Charniak. 1993. A language for construction of belief networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15 (3): 196-208.

[11] Horsch, M. C., and D. Poole. 1990. A dynamic approach to probabilistic inference using Bayesian networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, Cambridge, MA, pp. 27-9.

[12] Haddawy, P. 1994. Generating Bayesian networks from probability logic knowledge bases. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, San Francisco, pp. 262-9.

[13] Poole, D. 1993. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence* 64 (1): 81-129.

[14] Koller, D., and A. Pfeffer. 1997. Object-oriented Bayesian networks. In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, Providence, RI, pp. 302-13.

[15] Mahoney, S. M., and K. B. Laskey. 1998. Constructing situation specific belief networks. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, Wisconsin, USA, pp. 370-8.

[16] Gertner, A. S., C. Conati, and K. Vanlehn. 1998. Procedural help in andes: Generating hints using a Bayesian network student model. In *Proceedings of AAAI*, Menlo Park, CA, pp. 106-11.

[17] Bangso, O., and P. H. Wuillemin. 2000. Top-down construction and repetitive structures representation in Bayesian networks. In *Proceedings of the 13th International Florida Artificial Intelligence Research Society Conference*, Orlando, USA.

[18] Xiang, Y. 2002. *Probabilistic reasoning in multi-agent systems: A graphical models approach*. Cambridge, UK: Cambridge University Press.

Y. Xiang *is a professor in the Department of Computing and Information Science, College of Physical and Engineering Science, University of Guelph, Ontario, Canada.*

X. An *is a PhD candidate at the University of Waterloo, Canada.*

N. Cercone *is a professor and dean in Faculty of Computer Science at Dalhousie University, Canada.*