Verification of DAG Structures in Cooperative Belief Network Based Multi-agent Systems

Y. Xiang
Department of Computer Science, University of Regina
Regina, Saskatchewan, Canada S4S 0A2, yxiang@cs.uregina.ca

Abstract

Multiply sectioned Bayesian networks (MSBNs) provide a framework for probabilistic reasoning in a complex single user oriented system as well as in a cooperative multi-agent distributed interpretation system. During the construction or dynamic formation of a MSBN, automatic verification of the acyclicity of the overall structure is desired. Well known algorithms for acyclicity test assume a centralized storage of the structure to be tested. We discuss why a centralized test is undesirable and propose a distributed algorithm that verifies the acyclicity through cooperation among subnets/agents. The algorithm does not require each agent to reveal its internal structure and thus supports construction of a MSBN from subnets built by different vendors.

1 Introduction

Multiply sectioned Bayesian networks (MSBNs) is an extension of Bayesian networks (BNs) [1, 4, 3]. A MSBN consists of a set of interrelated Bayesian subnets that collectively define a BN [12, 11]. Each subnet shares a non-empty set of variables with at least one other subnet. Subnets are organized into a hypertree structure such that probabilistic inference can be performed coherently in a modular and distributed fashion. The modularity improves inference efficiency in a complex single user oriented system [10]. It also allows MSBNs to be extended into a coherent framework for probabilistic reasoning in cooperative multi-agent distributed interpretation systems [8].

The structure of a BN is a directed acyclic graph (DAG). The overall structure of a MSBN, the composition of subnet structures, is also a DAG. To ensure the correct composition, automatic verification of acyclicity of the composed structure is desired. Although algorithms for testing acyclicity based on topological sorting are well known, see [6] for example, they assume a centralized storage of the graph to be tested. We analyze some design considerations that make a centralized test undesirable. We then propose a distributed algorithm in terms of a set of distributed operations for testing acyclicity of the composed structure through cooperation of subnets/agents.

The theory and applications of MSBNs are briefly reviewed in Section 2. The concepts necessary to the rest of the paper are formally defined. We discuss in Section 3 reasons

why a distributed verification of acyclicity is preferred. It is shown in Section 4 that some 'obvious' solutions to distributed verification do not solve the problem. The graph-theoretic foundation for the proposed algorithm is derived in Section 5 and the algorithm is presented in Section 6 with a proof of its correctness. Its complexity is analyzed in Section 7.

2 Overview of MSBNs

In this section, we briefly overview the theory of MSBNs and their applications. More details on MSBNs can be found in [12, 10, 7, 8].

A BN S is a triplet (N, D, P) where N is a set of variables, D is a DAG whose nodes are labeled by elements of N, and P is a joint probability distribution (jpd) over N. We shall call N the *domain* of S, D the *structure* of S and P the *distribution* or jpd of S.

A MSBN M is a collection of Bayesian subnets that together define a BN. These subnets are required to satisfy certain conditions that permit the construction of distributed inference algorithms. One of these conditions requires that nodes shared by different subnets form a *d-sepset*, as defined below.

Let $G^i = (N^i, E^i)$ $(i = 1, 2)$ be two graphs. We shall refer to the graph $G = (N^1 \cup N^2, E^1 \cup E^2)$ as the *union* of G^1 and G^2, denoted by $G = G^1 \sqcup G^2$.

Definition 1 (d-sepset) *Let $D^i = (N^i, E^i)$ $(i = 1, 2)$ be two DAGs such that $D = D^1 \sqcup D^2$ is a DAG. The intersection $I = N^1 \cap N^2$ is a* `d-sepset` *between D^1 and D^2 if for every $A \in I$ with its parents π in D, either $\pi \subseteq N^1$ or $\pi \subseteq N^2$. Each node in a d-sepset is called a* `d-sepnode`*.*

The d-sepset concept is a *syntactic* condition. *Semantically*, it can be shown that when a pair of subnets are isolated from M, their d-sepset renders them conditionally independent. Therefore, d-sepset provides a simple syntactic rule to facilitate independent specification of semantically correct subnets. Figure 1 (left) shows the three DAGs D^i $(i = 1, 2, 3)$ of a MSBN for diagnosis of three neuromuscular diseases, Median nerve lesion (Medn), Carpal tunnel syndrome (Cts) and Plexus upper trunk lesion (Pxut).[1] The d-sepset between each pair of DAGs is $\{Medn, Cts, Pxut\}$. In general, d-sepsets between different pairs of DAGs of M may be different.

Just as the structure of a BN is a DAG, the structure of a MSBN is a multiply sectioned DAG (MSDAG) with a hypertree organization, or simply a *hypertree MSDAG* defined as follows:

Definition 2 (Hypertree MSDAG) *A hypertree MSDAG $\mathcal{D} = \bigsqcup_i D^i$, where each D^i is a connected DAG, is a DAG that is built by the following procedure:*

Start with an empty graph (no node). Recursively add a DAG D^k, called a `hypernode`*, to the existing MSDAG $\bigsqcup_{i=1}^{k-1} D^i$ subject to the constraints:*

[d-sepset] For each D^j $(j < k)$, the intersection $I^{jk} = N^j \cap N^k$ is a d-sepset when the two DAGs are isolated.

[1] The example is taken from a fraction of PAINULIM [10] modified for illustration.

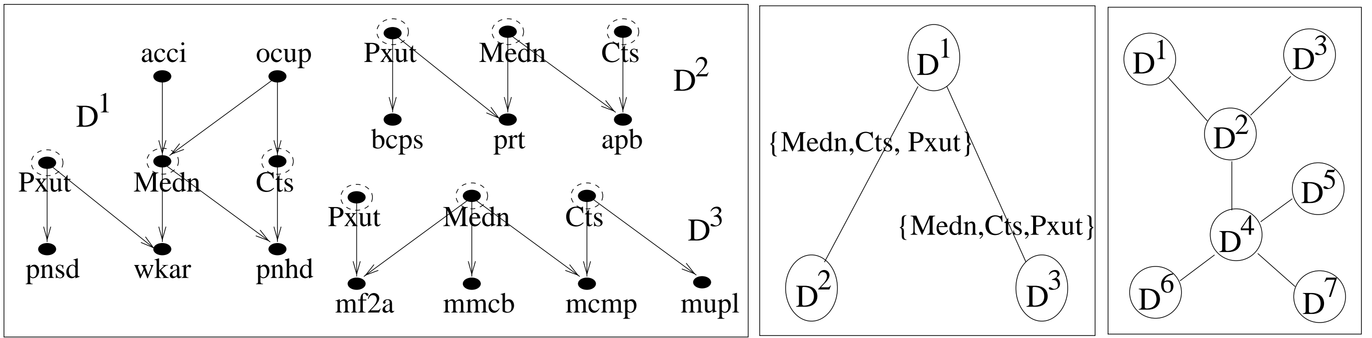

Figure 1: Left: The DAGs of an example MSBN, where each d-sepnode is highlighted by a dotted circle. Middle: The hypertree organization of the DAGs in the left. Right: A general hypertree MSDAG (unrelated to the left).

[Local covering] There exists D^i ($i < k$) such that, for each D^j ($j < k; j \neq i$), we have $I^{jk} \subseteq N^i$. For an arbitrarily chosen such D^i, I^{ik} is called the `hyperlink` *between hypernodes D^i and D^k, and D^i and D^k are said to be* `adjacent`.

Note that a hypertree MSDAG is a *tree* where each node is a hypernode as defined above and each link is a hyperlink. The DAGs in Figure 1 (left) can be organized into the trivial hypertree MSDAG in Figure 1 (middle), where each hypernode is labeled by a DAG and each hyperlink is labeled by a d-sepset. Figure 1 (right) depicts a general hypertree MSDAG. Although DAGs of a MSBN should be organized into a hypertree, each DAG may be multiply connected (more than one path exist between a pair of nodes), e.g., D^1. Moreover, there can be multiple paths between a pair of nodes in different DAGs in a hypertree MSDAG. For instance, multiple paths are formed between *apb* and *mcmp* after D^2 and D^3 are unioned. The local covering condition ensures that for any undirected cycle across two adjacent DAGs, both of its two paths are through the corresponding d-sepset. Together with the d-sepset condition, they ensure that in a hypertree structured M, each hyperlink renders the two parts of M that it connects conditionally independent. An intuitive justification of this structure is given in [9].

A MSBN is defined as follows. Readers are referred to [12] for more details.

Definition 3 *A MSBN M is a triplet $(\mathcal{N}, \mathcal{D}, \mathcal{P})$. $\mathcal{N} = \bigcup_i N^i$ is the* `total universe` *where each N^i is a set of variables. $\mathcal{D} = \bigsqcup_i D^i$ (a hypertree MSDAG) is the* `structure` *where nodes of each DAG D^i are labeled by elements of N^i. $\mathcal{P} = \prod_i P^i(N^i) / \prod_k P^k(I^k)$ is the* `joint probability distribution` *(jpd). Each $P^i(N^i)$ is a probability distribution over N^i such that whenever D^i and D^j are adjacent in $\mathcal{D}$, the marginalizations of $P^i(N^i)$ and $P^j(N^j)$ onto the d-sepset I^{ij} are identical. Each $P^k(I^k)$ is such a marginal distribution over a hyperlink of $\mathcal{D}$. Each triplet $S^i = (N^i, D^i, P^i)$ is called a* `subnet` *of M.*

Without confusion, we shall say that two subnets S^i and S^j are adjacent if D^i and D^j are adjacent.

A MSBN can be used as a framework for probabilistic reasoning in a single user oriented system in a large problem domain. The *single user* implies that evidence and queries are restricted to *one* subdomain at a time. Using a MSBN is most beneficial if subdomains of the problem domain are loosely coupled (the size of each d-sepset is reasonably small relative

to the size of the subdomain) and evidence and queries are focused on one subdomain for a period of time before shifting to a different subdomain. For example in Figure 1 (right), the user may focus attention on a subnet S^1 whose structure is D^1. After several pieces of evidence are entered and queries are issued to this subnet, the user may shift attention to the subnet S^3. The inference operations of MSBNs will then propagate evidence from S^1 to S^2 and then to S^3. The user can then enter evidence on variables contained in S^3. It can be shown that with such a restricted belief propagation during attention shift, the answers to queries obtained in S^3 are always consistent to all evidence accumulated in the *entire* MSBN. Computational complexity, however, is reduced by not having to update any subnets not on the hyperpath from the current subnet to the next target subnet. Application domains of single-user MSBNs include diagnosis of natural systems [10] and model-based diagnosis of artificial systems [5][2].

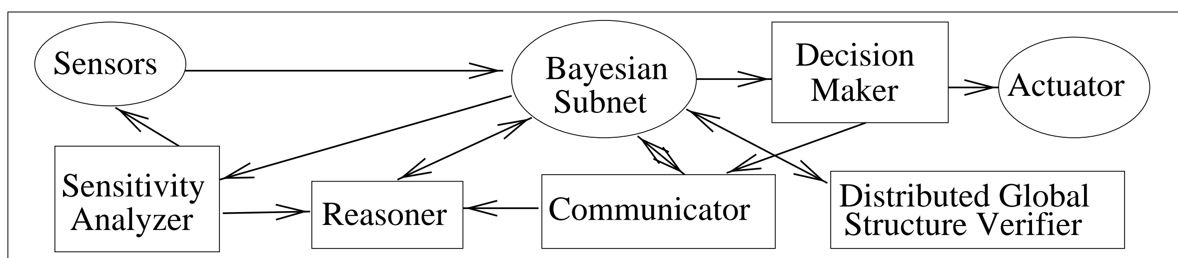

Figure 2: Main components of an agent in a MSBN-based multi-agent system.

MSBNs can be extended into a framework for probabilistic reasoning in cooperative multi-agent distributed interpretation systems. Each agent holds its partial perspective of a large problem domain (Subnet in Figure 2), accesses a local evidence source (Sensors in Figure 2), communicates with other agents *infrequently* (Communicator), reasons with the local evidence and limited global evidence (Reasoner), and answers queries (Reasoner) or takes actions (Decision Maker/Actuator). It can be shown [8] that if all agents are cooperative (vs self-interested), and each pair of adjacent agents are conditionally independent given their shared variables and have common initial belief on the shared variables, then a joint system belief is well defined which is identical to each agent's belief within its subdomain and supplemental to the agent's belief outside the subdomain. Even though multiple agents may acquire evidence asynchronously in parallel (compared with the single user oriented system where evidence is always entered into the current subnet of focus), the communication operations of MSBNs ensure that the answers to queries from each agent are consistent with evidence acquired in the entire system after each communication. Since communication is infrequent, the operations also ensure that between two successive communications, the answers to queries for each agent are consistent with all local evidence gathered so far and are consistent with all evidence gathered in the entire system up to the last communication. Therefore, a MSBN can be characterized as one of functionally accurate, cooperative distributed systems [2]. Potential applications include decision support to cooperative human

[2] Although MSBNs are not referenced directly, the representation formalism used is a special case of MSBNs. For example, the set of input nodes I, output node O, mode node M, and dummy node D [5], which forms an interface between a higher level and a lower level in the hierarchy, is a d-sepset [12]. The 'composite joint tree' [5] corresponds to the 'hypertree' [12]. The way in which inference is performed in the composite join tree corresponds to the operation *ShiftAttention* [12].

users in uncertain domains and troubleshooting a complex system by multiple knowledge based subsystems [8].

3 Why Distributed Verification?

As defined in Section 2, the structure of a MSBN is a hypertree MSDAG which should be a DAG. Automatic verification of acyclicity of this structure is desirable in the construction of large MSBNs. Algorithms that test whether a directed graph is a DAG based on topological sorting are well known, see for example [6]. These algorithms, however, assume a central representation of the graphical structure to be tested.

A central representation of all DAGs in a MSBN is not desirable for at least two reasons. First, the construction of a multi-agent MSBN requires only the knowledge of the functionality of each subnet and the interface (d-sepset) between subnets (BNs). Knowing the internal structure of each subnet is not necessary. Therefore, each subnet may be developed by an independent vendor who may not be willing to disclose the structural details. The assumption of a central representation of all DAGs will eliminate the possibility of cooperating agents built by such vendors.

Secondly, a MSBN can potentially be dynamic. That is, subnets may join or leave the MSBN as the system is functioning. It is desirable to verify the correctness of the structure of the system whenever the member subnets change. It is also desirable that the verification does not require the communication of all DAGs to a central location or does not depend upon a single agent to maintain a repository of all DAGs in the current system.

In this paper, we propose a distributed algorithm for verification of the acyclicity of a MSBN structure. During the verification process, each agent only provides answers to adjacent subnets on questions regarding d-sepnodes, and it does not reveal its internal structure beyond that.

4 Issues in Distributed Verification

Recall from Definition 2 that a hypertree MSDAG is built from a set of DAGs subject to the d-sepset and local covering conditions. However a directed graph built from a set of DAGs subject to these two conditions may still contain directed cycles. We shall refer to the resultant graph as a *hypertree DAG union* since it may not qualify as a hypertree MSDAG.

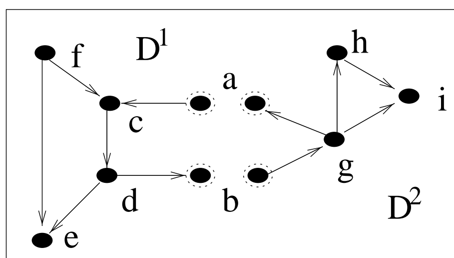

Figure 3: A cyclic DAG union.

Figure 3 shows two DAGs D^1 and D^2 with their d-sepset being $\{a, b\}$. If we union the two DAGs, it clearly satisfies the local covering condition. However, the union contains the directed cycle (a, c, d, b, g, a) and thus is not a DAG.

The above cycle can be detected if we union the pair of DAGs and test the acyclicity. Although the pairwise verification may detect some directed cycles, pairwise acyclicity in a hypertree DAG union does not guarantee the global acyclicity.

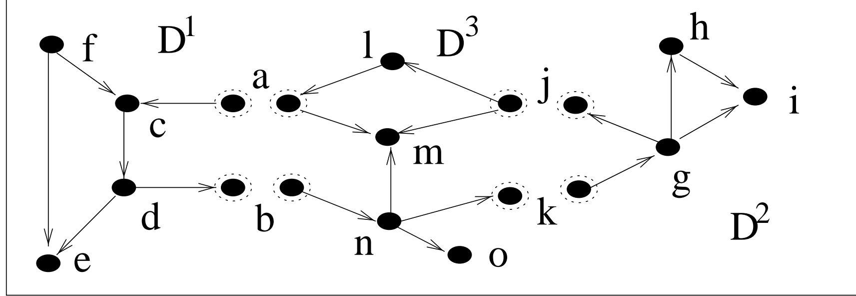

Figure 4: Three DAGs which are pairwise acyclic but whose union is cyclic.

Consider the three DAGs in Figure 4. The union of D^1 and D^3 is acyclic and so is the union of D^3 and D^2. However, when the three DAGs are unioned, a directed cycle $\{a, c, d, b, n, k, g, j, l, a\}$ is formed. Clearly, a distributed verification of acyclicity requires cooperation beyond pairs.

5 Verification by Marking Nodes

In this section, we show that acyclicity of a directed graph can be verified by marking root and leaf nodes recursively. Once it is established, we can mark non-d-sepnodes locally and mark d-sepnodes by cooperation as presented in the next section. A node is a *root* if each arc connected to it is directed away from it. A node is a *leaf* if each arc connected to it is directed toward it. A node x is *marked* if x and arcs connected to x are ignored from further verification process. The following two propositions show that marking of root/leaf nodes does not change acyclicity.

Proposition 4 *Let G be a directed graph and x be either a root or a leaf in G. Then the acyclicity of G remains after x is marked.*

Proof:

If G is acyclic, then marking x cannot create a directed cycle in G. Suppose G is cyclic. Then there exists a non-empty set O of directed cycles in G. If x is a root, it does not have any incoming arc. If x is a leaf, it does not have any outgoing arc. Therefore, x cannot participate in any cycles in O, which implies that none of the cycles in O will be changed after x is marked. $\square$

Once a root or leaf is marked, other nodes may become roots or leaves. Hence marking roots and leaves can be performed recursively while preserving the acyclicity.

Next, we show that if a directed graph is acyclic, every node in it will be marked by recursive applications of Proposition 4. On the other hand, if it is cyclic, at least three nodes will be left unmarked.

Proposition 5 *Let G be a directed graph. G is acyclic iff it is empty after recursive marking of roots and leaves.*

Proof:

Without losing generality, we assume that G has at least two nodes and is connected. Suppose G is acyclic. Then G has at least one root and one leaf. According to Proposition 4, after all of them are marked, the resultant graph is still acyclic and has new roots and leaves. Since G has a finite number of nodes, after recursive marking of roots and leaves, eventually G will have no unmarked nodes.

Next, suppose G is cyclic. Then G has at least one directed cycle θ consisting of at least three nodes. For each node x in θ, it is neither a root nor a leaf and thus cannot be marked as such. Marking of any nodes outside θ cannot turn x into a root or a leaf. Hence none of the nodes in θ can be marked by recursive marking of roots and leaves. Since G has a finite number of nodes, after recursive marking of roots and leaves, eventually there will be no roots or leaves to mark in G while all nodes (at least three) in θ are unmarked. □

We now consider a hypertree DAG union G whose nodes are classified into d-sepnodes and non-d-sepnodes. Since G is a connected directed graph, Proposition 5 can be applied to test its acyclicity. However, although non-d-sepnode roots and leaves can be recognized locally within each subnet, d-sepnode roots and leaves can only be recognized through cooperation among subnets. For example, the node i (a non-d-sepnode) in Figure 4 is a leaf both in D^2 (appearing markable locally) and in the DAG union (hence markable globally). On the other hand, k (a d-sepnode) is a leaf in D^3 (appearing markable locally), a root in D^2 (appearing markable locally), but a non-root/non-leaf in the DAG union (hence not markable globally). Moreover, marking of d-sepnode roots and leaves may turn some non-d-sepnodes into new roots or leaves. The following proposition shows that recursive and alternate marking of non-d-sepnode roots/leaves and d-sepnode roots/leaves is sufficient to test the acyclicity of a hypertree DAG union.

Corollary 6 *Let G be a hypertree DAG union. Let G' be the graph resulting from recursive and alternate marking of non-d-sepnode roots/leaves and d-sepnode roots/leaves in G until no more nodes can be marked. Then G is acyclic iff G' is empty.*

Proof:

According to Proposition 5, if G is acyclic, at each round of recursive marking, either some non-d-sepnode roots/leaves or some d-sepnode roots/leaves can be marked, until G is empty. If G is cyclic, at each round of recursive marking, either some non-d-sepnode roots/leaves or some d-sepnode roots/leaves can be marked, until only nodes in directed cycles in G are left unmarked (at least three).

The alternate marking of d-sepnodes and non-d-sepnodes is necessary. Otherwise, the marking may halt prematurely even if G is acyclic. □

To illustrate the necessity of alternate marking, consider the acyclic DAG union in Figure 5. Without using alternate marking, we have only two options: (a) recursive marking of

all non-d-sepnode roots/leaves followed by recursive marking of all d-sepnode roots/leaves, or (b) recursive marking of all d-sepnode roots/leaves followed by recursive marking of all non-d-sepnode roots/leaves. Using option (a), non-d-sepnodes c (root) and d (leaf) in D^1 will be marked in the first stage. Neither of the non-d-sepnodes e and f in D^2 can be marked at this stage. In the second stage, the d-sepnodes a (now a root) and b (now a leaf) can be marked. The marking terminates with e and f unmarked. Using option (b), no d-sepnode can be marked in the first stage since neither a nor b is a root or leaf. In the second stage, non-d-sepnodes c (root) and d (leaf) in D^1 will be marked. The marking terminates with a, b, e and f unmarked. Using alternate marking by starting with non-d-sepnodes, c and d will be marked in the first stage. In the second stage, d-sepnodes a and b will be marked. In the third stage, non-d-sepnodes e and f will be marked. Now the resultant graph is empty. Alternate marking by starting with d-sepnodes gives the same result.

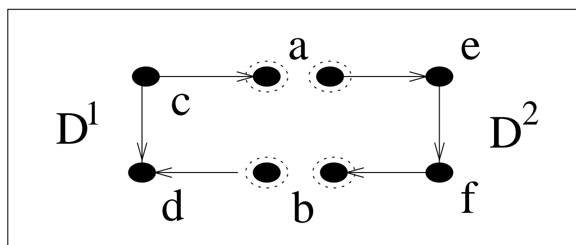

Figure 5: A cyclic DAG union.

Note that the corollary holds even when G is an arbitrary DAG union (not subject to the two conditions in Definition 2), and *d-sepnodes* are replaced by *shared nodes*. Such generality is not needed for our purpose.

Corollary 6 forms the basis for a distributed verification algorithm which we present in Section 6.

6 Cooperative Verification

As demonstrated in Sections 4 and 5, in order to verify the acyclicity of a hypertree DAG union, agents must cooperate. Since cooperation requires communication which incurs overhead, it is desirable to simplify the task for cooperation as much as possible. According to Corollary 6, non-d-sepnode roots/leaves in a hypertree DAG union can be marked separately and recursively. We define a preprocessing operation to mark these nodes before cooperation starts.

Let a DAG in a hypertree DAG union G be arbitrarily chosen. If we treat this DAG as the root of the hypertree and direct the hyperlinks of the hypertree away from it, then the hypertree is converted into a *directed* tree. For each given DAG, we can then refer to each adjacent DAG as its *child* or its *parent* in the normal sense.

Operation 7 (PreProcess) *When* `PreProcess` *is called in a DAG D, the following are performed:*

1. *D recursively marks each non-d-sepnode root or leaf.*

2. *D calls* `PreProcess` *in each child DAG.*

After `PreProcess` is completed in G, nodes left unmarked in each DAG are either isolated d-sepnodes, or nodes that form directed paths ended with d-sepnodes. Cooperation among DAGs is needed to further the verification process. Figure 6 shows the three DAGs in Figure 4 after `PreProcess` is initiated in any of them. Only directed paths are left in this case. We will see isolated d-sepnodes in a later example.

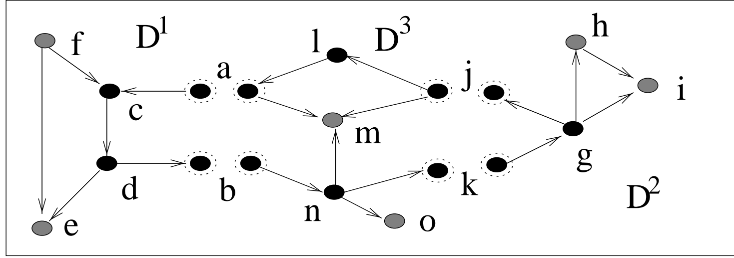

Figure 6: The DAG union in Figure 4 after local preprocessing. Marked nodes are shown as grey.

To find out if a d-sepnode x can be marked, a DAG uses the operation `CollectFamilyInfo` to determine if x is a root or leaf through cooperation. The operation passes a triple (x, p, c) around all child DAGs which contain x. The purpose is to collect the parent/child information for x, where p is a count of the number of DAGs that contain parents of x and c is a count of the number of DAGs that contain children of x. The caller in the following definition refers to either a parent DAG or the next higher level of operation which initiated this operation.

Operation 8 (CollectFamilyInfo) *When* `CollectFamilyInfo(x)` *is called in a DAG D, the following are performed:*

1. *D forms a triple $t_0 = (x, p_0, c_0)$, where $p_0 = 1$ if D contains an (unmarked) parent of x and $p_0 = 0$ otherwise, and $c_0 = 1$ if D contains an (unmarked) child of x and $c_0 = 0$ otherwise.*

2. *If D has no child DAG to which x is a d-sepnode, or $p_0 = c_0 = 1$, then D returns t_0 to* `caller`*.*

3. *Otherwise, D calls* `CollectFamilyInfo(x)` *in each child DAG to which x is a d-sepnode.*

4. *After each child DAG being called has returned their triples (assuming k DAGs are called), $t_1, t_2, \ldots, t_k$, D returns a triple $t = (x, p = \max_{i=0}^{k} p_i, c = \max_{i=0}^{k} c_i)$ to* `caller`*.*

Once a d-sepnode x is determined to be a root ($p = 0$) or a leaf ($c = 0$), the operation `DistributeMark` is used to mark it in every DAG that contains it.

Operation 9 (DistributeMark) *When* `DistributeMark(x)` *is called in a DAG D, the following are performed:*

1. *D marks the node x.*

2. *D recursively marks any non-d-sepnode root or leaf.*

3. *If D has any adjacent DAG to which x is a d-sepnode except* `caller`*, then D calls* `DistributeMark(x)` *in each of them.*

Note that in this operation the marking of each d-sepnode is alternated with the marking of non-d-sepnodes as required by Corollary 6.

The operation `MarkNode` combines `CollectFamilyInfo` and `DistributeMark` to perform one round of marking of d-sepnodes. It marks each d-sepnode root/leaf down the hypertree, and alternates each marking with the marking of non-d-sepnode roots/leaves in each DAG.

Operation 10 (MarkNode) *When* `MarkNode` *is called in a DAG D, the following are performed:*

1. *D returns* `false` *if it has no child DAG, otherwise continues.*

2. *For each unmarked d-sepnode x with a child DAG of D, D calls* `CollectFamilyInfo(x)` *in itself. When the triple (x, p, c) is returned to D, D calls* `DistributeMark(x)` *in itself if $p = 0$ or $c = 0$.*

3. *D calls* `MarkNode` *in each child DAG.*

4. *If any child DAG returns* `true` *or* `DistributeMark(x)` *was called in D, then D returns* `true` *to* `caller`*. Otherwise, D returns* `false` *(no node is marked).*

The operation `MarkedAll` checks whether all nodes in a hypertree DAG union have been marked after roots and leaves have been recursively marked. According to Corollary 6, G is acyclic iff `true` is returned.

Operation 11 (MarkedAll) *When* `MarkedAll` *is called in a DAG D, the following are performed:*

1. *If there exists a node in D that has not been marked, then D returns* `false`*.*

2. *Otherwise, if D has no child DAG, it returns* `true`*. If D has child DAGs, D calls* `MarkedAll` *in each child DAG.*

3. *If any child DAG returns* `false` *(with unmarked nodes), then D returns* `false`*. Otherwise, D returns* `true`*.*

Finally, the top level operation `TestAcyclicity` combines the previously defined operations to verify the acyclicity of G.

Operation 12 (TestAcyclicity) *When* `TestAcyclicity` *is initiated in a hypertree DAG union G, the following are performed:*

1. *A DAG D is arbitrarily chosen as the root of the hypertree.*
2. *D calls* `PreProcess` *in itself.*
3. *D calls* `MarkNode` *in itself repeatedly until* `false` *is returned (no node is marked in the last call).*
4. *D calls* `MarkedAll` *in itself. If* `true` *is returned, then* `TestAcyclicity` *returns* `acyclic` *(G is acyclic). Otherwise, return* `cyclic`*.*

The following theorem establishes the correctness of the algorithm.

Theorem 13 *The operation* `TestAcyclicity` *correctly determines the acyclicity of a hypertree DAG union.*

Proof:
According to Corollary 6, it is sufficient to mark non-d-sepnode roots/leaves and d-sepnode roots/leaves recursively and alternately. `PreProcess` does the first round of recursive marking of non-d-sepnode roots/leaves, and repeated `MarkNode` performs the subsequent recursive and alternate marking. Each `MarkNode` identifies d-sepnode roots/leaves by `CollectFamilyInfo` (either $p = 0$ or $c = 0$) and then marks them as well as new non-d-sepnode roots/leaves by `DistributeMark`. By Corollary 6, `MarkNode` will not return `false` until all roots and leaves are marked. `MarkedAll` tests if the DAG union is empty and will determine the acyclicity correctly. □

We illustrate the performance of `TestAcyclicity` with two examples, a cyclic DAG union and an acyclic one. The first is the DAG union depicted in Figure 4.

- Suppose D^1 is selected as the root. After `PreProcess` the union looks as Figure 6.
- When D^1 calls `MarkNode` in itself, it calls in itself `CollectFamilyInfo(a)` which is then propagated to D^3. D^3 returns a triple $(a, 1, 0)$ to D^1. Subsequently, D^1 generates the final triple $(a, 1, 1)$ and terminates `CollectFamilyInfo(a)`. Since neither p nor c is zero, `DistributeMark(a)` is not called.

  D^1 then calls in itself `CollectFamilyInfo(b)` which eventually terminates similarly as `CollectFamilyInfo(a)`.
- D^1 calls `MarkNode` in D^3. D^3 calls in itself `CollectFamilyInfo(j)` which is then propagated to D^2. D^2 returns a triple $(j, 1, 0)$ to D^3. Subsequently, D^3 generates the final triple $(j, 1, 1)$ and terminates `CollectFamilyInfo(j)`. No `DistributeMark` is called.

  D^3 then calls in itself `CollectFamilyInfo(k)` which eventually terminates similarly as `CollectFamilyInfo(j)`.
- D^3 calls `MarkNode` in D^2 which returns `false` since D^2 has no child DAG. This causes D^3 to return `false` to D^1 which also returns `false` and terminates `MarkNode`.
- D^1 calls `MarkedAll` in itself and returns `false` immediately. `TestAcyclicity` then terminates with `cyclic` returned.

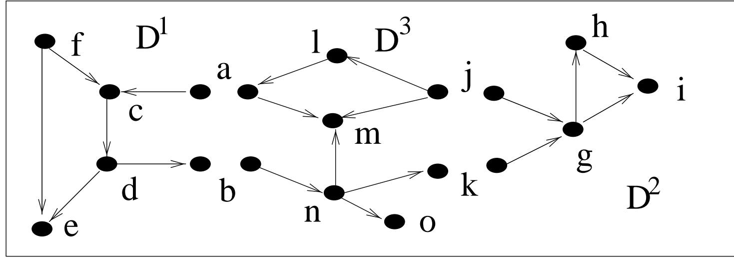

Figure 7: An acyclic DAG union.

As another example, consider the DAG union in Figure 7. It is identical to that in Figure 4 except that the arc from g to j is now reversed.

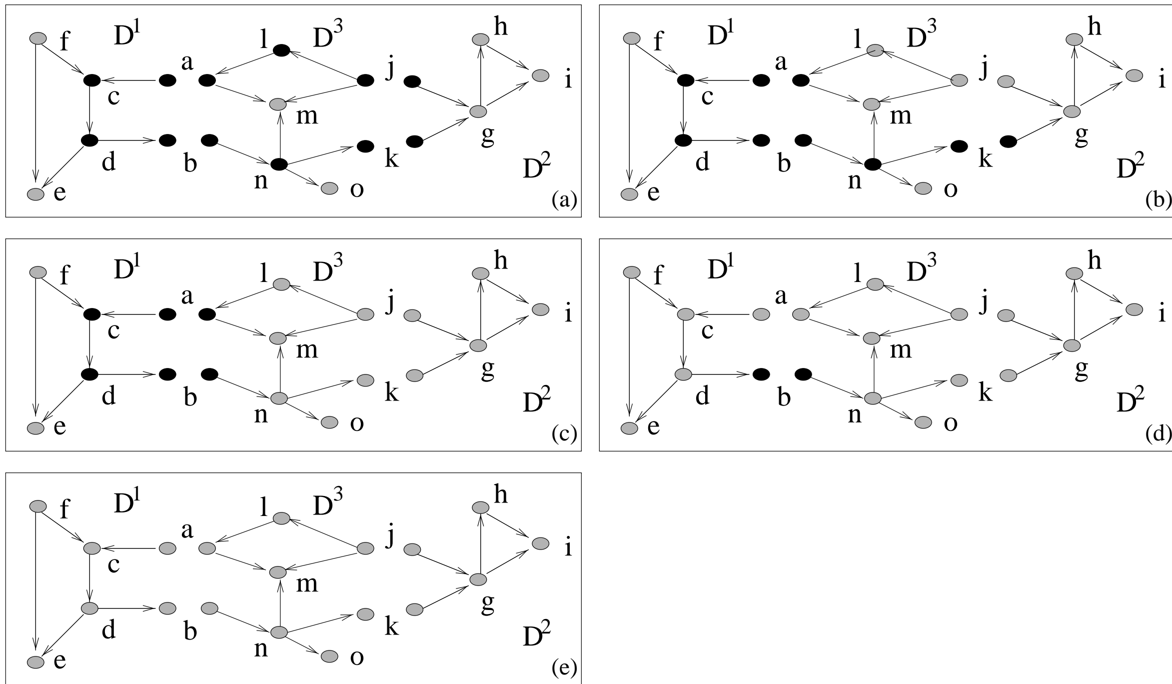

Figure 8: Performance of `TestAcyclicity` in the DAG union of Figure 7.

- Suppose D^1 is selected as the root. Figure 8 (a) shows the union after `PreProcess`. Note that nodes j and k are isolated in D^2 but cannot be marked during `PreProcess`. `PreProcess` is so defined since an isolated d-sepnode in one DAG may still participate in a directed cycle in other DAGs. It would be inconsistent to mark it in one DAG and to keep it unmarked in another.
- When `MarkNode` is first called in D^1 within `TestAcyclicity`, it calls `CollectFamilyInfo(a)` in itself and then `CollectFamilyInfo(b)` as in the previous example.

- D^1 calls `MarkNode` in D^3. D^3 calls in itself `CollectFamilyInfo(j)` which is then propagated to D^2. D^2 returns a triple $(j, 0, 0)$ to D^3. Subsequently, D^3 generates the final triple $(j, 0, 1)$ and terminates `CollectFamilyInfo(j)`.

  Since $p = 0$, D^3 calls `DistributeMark(j)` in itself. It marks j and l, and then calls `DistributeMark(j)` in D^2 which marks j as well. Figure 8 (b) shows the resultant union.

  D^3 then calls in itself `CollectFamilyInfo(k)` which eventually terminates similarly as `CollectFamilyInfo(j)`. Figure 8 (c) shows the resultant union.

- D^3 calls `MarkNode` in D^2 which returns `false`. Since `DistributeMark` was called in D^3, it returns `true` to D^1 which returns `true` and terminates the first call of `MarkNode` in `TestAcyclicity`.

- When `MarkNode` is called in D^1 the second time within `TestAcyclicity`, D^1 calls in itself `CollectFamilyInfo(a)` which is then propagated to D^3. D^3 returns a triple $(a, 0, 0)$ to D^1. Subsequently, D^1 generates the final triple $(a, 0, 1)$ and terminates `CollectFamilyInfo(a)`.

  Since $p = 0$, D^1 calls `DistributeMark(a)` in itself. This causes the marking of a, c, d in D^1 and a in D^3. Figure 8 (d) shows the resultant union.

  D^3 then calls in itself `CollectFamilyInfo(b)`. It eventually terminates similarly as `CollectFamilyInfo(a)` with b marked in both D^1 and D^3. Figure 8 (e) shows the resultant union.

- D^1 calls `MarkNode` in D^3 which then calls `MarkNode` in D^2. D^2 has no child DAG and returns `false`, which causes D^3 to return `false` to D^1. Since `DistributeMark` was called in D^1, it returns `true` and terminates the second call of `MarkNode` in `TestAcyclicity`.

- When `MarkNode` is called in D^1 the third time within `TestAcyclicity`, it propagates `MarkNode` to D^3 and then to D^1. Eventually, `false` is returned.

- When `MarkedAll` is called in D^1, it propagates the operation to the rest of the union. Eventually, `true` is returned.

- The operation `TestAcyclicity` terminates with `acyclic` returned.

7 Complexity Analysis

We denote the maximum number of nodes in a DAG by m, the maximum number of adjacent nodes of a node in a DAG by t, the maximum number of nodes in a d-sepset by k, the maximum number of DAGs that may contain a d-sepnode by s, and the total number of DAGs in the hypertree DAG union by n.

To recursively mark all non-d-sepnodes in a DAG, $O(t\ m)$ nodes need to be checked. Hence `PreProcess` checks $O(n\ t\ m)$ nodes.

Each `CollectFamilyInfo` (relative to a single d-sepnode) tests a d-sepnode in $O(s)$ DAGs. To mark a d-sepnode, $O(s)$ DAGs perform the marking, each of which also checks

$O(t\ m)$ non-d-sepnodes. Hence the complexity of `CollectFamilyInfo` and `DistributeMark` for each *marked* d-sepnode is $O(s\ t\ m)$. The complexity of each `MarkNode` called from `TestAcyclicity` is then $O(n\ k\ s\ t\ m)$. Since at least one d-sepnode will be marked for each call of `MarkNode`, `Marknode` will be called $O(n\ k)$ times. Hence the complexity of all `MarkNode` calls is $O(n^2\ k^2\ s\ t\ m)$.

`MarkedAll` checks $O(n\,m)$ nodes. Therefore, the worst case complexity of `TestAcyclicity` is $O(n^2\ k^2\ s\ t\ m)$.

8 Discussion

In this paper, we presented an efficient distributed algorithm, for verification of acyclicity of the overall structure of a MSBN. An important feature of the algorithm is that it does not require each subnet/agent in the system to reveal its internal structure. From the definition of `CollectFamilyInfo`, clearly each agent only provides information regarding whether each shared node has any parent or child in the DAG that the agent is responsible for. Therefore, the algorithm supports the construction of MSBNs constructed from multiple computational agents built by multiple vendors while providing automatic verification of correctness of the overall structure.

Our distributed algorithm is based on Corollary 6 which in turn is based on Proposition 5. Proposition 5 extends the idea of topological sorting in that the latter is equivalent to marking only root nodes. Even in a centralized test, Proposition 5 allows a more efficient test than topological sorting. This is because Proposition 5 admits on average twice as many markable nodes at each recursive marking, leaving less nodes to check in subsequent marking processes. Although topological sorting can also be extended into a distributed algorithm, the advantage of `TestAcyclicity` is more prominent in cooperative test. Since `MarkNode` marks more nodes in each round than topological sorting does, many less calls of `MarkNode` will be made (much more efficient cooperation), which translates into a reduction in communication overhead.

As mentioned in Section 3, distributed verification facilitates dynamic formation of a MSBN. In general, `TestAcyclicity` should be performed whenever the member subnets of a MSBN change. However, there are special cases where the execution of a full scale `TestAcyclicity` is unnecessary. For example, if a new subnet only interfaces with one existing subnet and the d-sepset between them contains root nodes only, then the acyclicity of the new DAG union can be confirmed locally. One useful direction for future research is to identify such special cases and to develop simplified verification operations accordingly.

Acknowledgements

This work is supported by the Research Grant OGP0155425 from the Natural Sciences and Engineering Research Council (NSERC) of Canada. Helpful comments on an earlier draft from T. Chu and S. Hunter are acknowledged.

References

[1] E. Charniak. Bayesian networks without tears. *AI Magazine*, 12(4):50–63, 1991.

[2] V.R. Lesser and D.D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Trans. on Systems, Man and Cybernetics*, SMC-11(1):81–96, 1981.

[3] R.E. Neapolitan. *Probabilistic Reasoning in Expert Systems.* John Wiley and Sons, 1990.

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

[5] S. Srinivas. A probabilistic approach to hierarchical model-based diagnosis. In *Proc. 10th Conf. Uncertainty in Artificial Intelligence*, pages 538–545, Seattle, Washington, 1994.

[6] D.F. Stubbs and N.W. Webre. *Data Structures with Abstract Data Types and Modula-2.* Brooks/Cole, 1987.

[7] Y. Xiang. Optimization of inter-subnet belief updating in multiply sectioned Bayesian networks. In *Proc. 11th Conf. on Uncertainty in Artificial Intelligence*, pages 565–573, Montreal, 1995.

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

[9] Y. Xiang. Semantics of multiply sectioned Bayesian networks for cooperative multi-agent distributed interpretation. In G. McCalla, editor, *Advances in Artificial Intelligence*, pages 213–226. Springer, 1996.

[10] Y. Xiang, B. Pant, A. Eisen, M. P. Beddoes, and D. Poole. Multiply sectioned Bayesian networks for neuromuscular diagnosis. *Artificial Intelligence in Medicine*, 5:293–314, 1993.

[11] Y. Xiang, D. Poole, and M. P. Beddoes. Exploring locality in Bayesian networks for large expert systems. In *Proc. 8th Conf. on Uncertainty in Artificial Intelligence*, pages 344–351, Stanford, 1992.

[12] Y. Xiang, D. Poole, and M. P. Beddoes. Multiply sectioned Bayesian networks and junction forests for large knowledge based systems. *Computational Intelligence*, 9(2):171–220, 1993.