Distributed Multi-agent MSBN: Implementing Verification

Hongyu Geng and **Yang Xiang**
University of Regina, Regina, Saskatchewan, Canada

Abstract

Multiply Sectioned Bayesian Networks (MSBN) provide a coherence framework for multi-agent distributed interpretation tasks. During the construction or dynamic formation of an MSBN, automatic verification of d-sepset and the acyclicity of the overall structure is desired. Although verification has been implemented in a time-sharing fashion, new issues must be resolved in order to perform such verification of an MSBN in a distributed environment. We discuss these issues and algorithms for verification in a distributed environment.

Introduction

Bayesian networks (BNs) (Pearl 1988, Jensen 1996), as commonly applied, assume a single agent paradigm. Multiply sectioned Bayesian networks (MSBNs) are an extension of BNs to multi-agent systems (Xiang et al. 1993, Xiang 1996). An *MSBN* is a collection of interrelated BNs organized into a *hypertree* structure where each BN forms the core knowledge of an agent. An MSBN allows multiple agents to reason distributively about a large uncertain domain. Figure 1 shows the subnets of an MSBN.

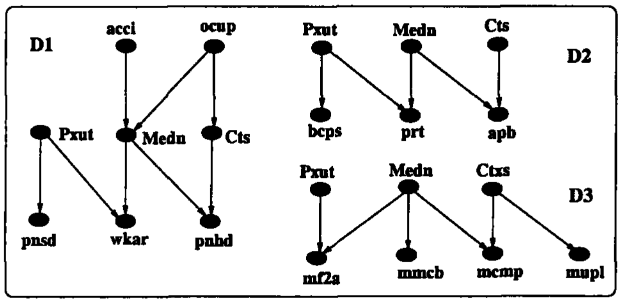

Figure 1: An example of MSBN for neural muscular diagnosis.

To ensure correct inference, subnets need be organized under certain conditions. The focus of this paper is on automatic verification of two such conditions and their implementation in a distributed environment. One is the *d-sepset* condition on nodes that interface a pair of subnets (Xiang 1996). Anther is the acyclicity of the union of all the subnets (Xiang 1998). These conditions should be verified whenever a multi-agent MSBN system is created or modified.

Roles of Agents

Previous implementation of the verification in WEBWEAVR (a research toolkit for normative decision support systems) is for a centralized environment. In this work, we extend to a distributed environment, where each agent/subnet is run on a different machine. To perform verification, agents must communicate with each other through a computer network.

Verification may be initiated by any agent who acts as a *root* of the hypertree. Other agents will respond by communicating with neighbors along the hypertree through socket interfaces. Using the common client-server model of socket, each agent needs a server to receive messages from potential callers and a client to send messages to a neighbor.

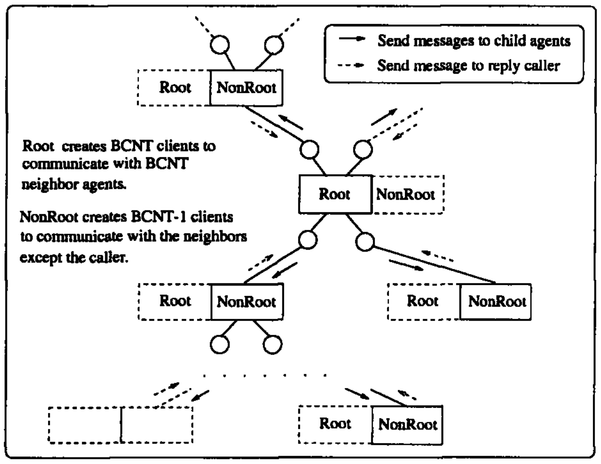
Send messages to child agents
Send message to reply caller
Root
NonRoot
Root creates BCNT clients to communicate with BCNT neighbor agents.
NonRoot creates BCNT-1 clients to communicate with the neighbors except the caller.

Figure 2: Each agent receives a message from its caller, and creates multiple clients to communicate with its child agents, then replies to the caller.

We distinguish the roles of *Root* and *NonRoot* an agent may play. If an agent initiates communication, possibly upon user's request, it works as a Root, otherwise the agent works as a NonRoot. For each execution of verification, only a single agent plays the role of Root, but different agents may act as the Root over different executions. When an agent performs the role of a *NonRoot*, there is another neighbor agent, acting as either Root or NonRoot, which requests communication with this agent. We call the requesting agent a "caller". A NonRoot agent must reply to its caller after it finishes its actions. Figure 2 illustrates these two roles.

Verify d-sepset

The d-sepset condition (Xiang 1996) ensures that variables shared by two subnets can pass *relevant* information at all cases. It prevents a shared variable to have non-shared parents that are split into two subnets. The d-sepset between each pair of DAGs of the MSBN in Figure 1 is {*Medn, Cts, Pxut*}. The verification is performed by the *TestDsepset* algorithm which calls the algorithm *CollectPaInfo*.

Algorithm 1 (CollectPaInfo)

```
/* collect a given variable's parents information */
input: caller index, a given variable v
begin
    collect v's parent information locally (result is outcnt);
    if this agent is not a leaf agent
        for each child agent
            create a message client to request the child to
            perform CollectPaInfo ;
        add result from each child agent to outcnt;
    return outcnt;
end
```

CollectPaInfo collects parent information *outcnt* of a given variable v. First, the host agent performs an operation locally to get the parent information of v in itself. If v has *out-parents* (parent nodes not in any sepset) locally, then $outcnt = 1$. Otherwise, $outcnt = 0$. Second, the host agent requests all child agents to get the parent information of v in each child agent. Finally, the result from each child agent is added to *outcnt* and then *outcnt* is returned. As a result, if $outcnt > 1$, then the given variable v has at least two out-parents, each of which belongs to a different agent.

TestDsepset verifies if all subnet interfaces are d-sepsets. For any sepnode, if its out-parents exist in more than one agent, then the sepset which contains this sepnode is not a d-sepset. Initially, the host agent collects the parent information of a sepnode v which is in the union of all intersections with all of the child agents. If the *outcnt* of a sepnode v is larger than 1, then the out-parents of v exist in more than one agent. Consequently, the sepset with the sepnode v is not a d-sepset and there is no need to continue. The *flag* is set to be false. Otherwise, if all the sepnodes in the union

Algorithm 2 (TestDsepset)

```
/* test if all sepset are d-sepsets. */
input: caller index
begin
    set flag = true;
    union sepsets with all neighbor agents;
    for each sepnode x in the union with a child agent
        collect out-parent info of x locally;
        if this agent is a leaf agent
            if outcnt > 1
                flag = false;
        else
            for each child agent
                create a message client to request the child
                agent to perform CollectPaInfo to collect
                parent info of x;
            add the result from each child to outcnt;
            if outcnt > 1
                flag = false;
            if flag = true
                for each child agent
                    create a message client to request the
                    child agent to perform TestDsepset;
                and each result from each child with flag;
    return flag;
end
```

have out-parents in only one agent, the host agent will request each child agent to perform *TestDsepset*. The result from each child agent will be "anded" with the *flag*. Finally, the value of *flag* is returned.

Verify Acyclicity

The union of all DAGs in a MSBN should be acyclic. Xiang (1998) presents a set of operations to verify the acyclicity. Here we reinterpret these operations under a distributed setting. A node in a directed graph is a *root* if it has no parent, otherwise a *leaf* if it has no child. Verification is based on node marking. If the union is acyclic, all nodes can be marked one by one as root or leaf. Otherwise, nodes which form a cycle are left unmarked as the end.

Four operations, *PreProcess*, *MarkNode*, *MarkedAll*, and *TestAcyclicity*, collectively verify acyclicity.

When *PreProcess* is performed, the host agent only marks the root or leaf non-d-sepset nodes, and then asks each child agent to do the same.

The family information of a variable in *CollectFamilyInfo* refers to the position of the node corresponding to the variable in the graph which consists of all subnets. When this algorithm is performed, host agent will collect the family information of the given variable

Algorithm 3 (PreProcess)

```
Input: caller index
begin
    mark non-d-sepset nodes that are root or leaf.
    if it has child agents
        for each child agent
            create a message client to request the child
            perform PreProcess;
end
```

Algorithm 4 (CollectFamilyInfo)

```
/* collect the family information of the given variable */
Input: caller index, variable name v.
begin
    collect v's family info locally;
    if this agent is not a leaf agent
        if node v is a root node or leaf node
            for each child agent
                create a message client to request the child
                to perform CollectFamilyInfo;
        handle v's family info with the collect result from
        each child agent;
    return v's family info;
end
```

v locally. Then, host agent will request its child agents to collect the family information of v if v is a root node or leaf node locally. Host agent will handle the result from each child agent with the local information and return the final result to its caller.

Algorithm 5 (DistributeMarkNode)

```
/* caller is always a neighbor agent */
Input: caller index, variable name v.
begin
    if v is a d-sepset node of host agent
        mark the given d-sepset node v locally;
        mark all non-d-sepset root node and leaf node;
    if this agent is not a leaf agent
        for each child agent
            create a message client to request the child
            agent to perform DistributeMarkNode;
end
```

When *DistributeMarkNode* is performed, host agent will mark the given variable v. New root nodes and new leaf nodes might be produced because of the marking of v. Therefore, host agent will mark the new non-d-sepset root node and leaf node. If this host agent is not a leaf agent, it will request its child agents to perform *DistributeMarkNode*.

MarkNode can be called by the system or the neighbor agents. First, *markCount*, a counter, is set to

Algorithm 6 (MarkNode)

```
/* mark each node when possible */
Input: caller index.
Output: markCount-the number of d-sepnodes marked
in the last round.
begin
    markCount = 0;
    union d-sepsets with all neighbors;
    for each unmarked d-sepnode x in the union
        if x is a d-sepnode with caller
            continue;
        call CollectFamilyInfo to collect the family
        information of x;
        if x is neither root node nor leaf node
            continue;
        else /* x is a root or leaf in the system */
            markCount ++;
            DistributeMarkNode is performed;
    for each child agent
        create a message client to request the child agent
        to perform MarkNode;
        add the result from each agent to markCount;
    return markCount;
end
```

count how many nodes will be marked after *MarkNode* is performed. Second, the d-sepsets with all neighbor agents except the caller are unioned. Third, for each unmarked d-sepnode v in the union, host agent performs *CollectFamilyInfo* to collect the family information of v. Fourth, if v is a root node or leaf node in the system, *DistributeMarkNode* is performed and *markCount* is incremented by one. Finally, if this host agent is not a leaf agent, it will send a message to each child agent, requesting each child agent to perform *MarkNode*. Each child agent will reply to the host agent with its *markCount* value.

Algorithm 7 (MarkedAll)

```
/* Check if there is any local/remote node unmarked. */
Input: caller index.
begin
    flag = true;
    if there is local node unmarked, flag = false;
    else if this agent is a leaf agent, flag = true;
        else send a message to each child agent
            to request each child to perform MarkedAll;
        process the flag with the result from each child;
    return flag;
end
```

When *MarkedAll* is performed, the host agent first sets *flag* to be *true*; If there is any unmarked node locally, then $flag = false$, and host agent will reply to its caller with the value "false". Otherwise, it will request its child agents to perform *MarkedAll*. If any one child replies to it with a "false" value, the *flag* of the host agent is false. Finally, host agent will reply to its caller with the value of its flag.

Algorithm 8 (TestAcyclicity)

```
/*Host agent test if there is a cycle in the union of all
subnets in the system. */
Input: caller index, message type
begin
   /* PreProcess phase */
   if msgType == PREPROCESS
      agent perform PreProcess;
   /* MarkNode phase */
   else if msgType == MARKNODE
      do { /* NonRoot only performs one round */
         agent perform MarkNode and return a value
         of markCount;
         if this agent is not a Root agent
            markCount = 0;
      } while (markCount != 0);
   /* MarkedAll phase */
   else if msgType == MARKEDALL
      agent performs MarkedAll;
end
```

The top-level algorithm *TestAcyclicity* combines the others to verify the acyclicity. *msgType* here stands for the type of message that the host agent receives from its caller. The host agent can be called by system or one of its neighbor agents to perform this algorithm. This algorithm is performed differently by an agent depending on whether this agent acts as a Root or NonRoot. The differences are two-folded. Firstly, if the host agent is a Root, the system will call the host agent to perform the three phases of the algorithm, *PreProcess*, *MarkNode* and *MarkedAll*, sequentially to perform this algorithm. Otherwise, the host agent will perform one phase of the algorithm once according to the message type. Secondly, when *MarkNode* phase is performed, Root will repeatedly perform *MarkNode* until there is no nodes to be marked in the last round (markCount = 0). NonRoot agent will only perform *MarkNode* for one round when it receives the request and will return the value of *markCount* to its caller.

The complexity of *TestAcyclicity* is polynomial on the number of subnets as well as the size of each subnet (Xiang 1998). Our distributed implementation does not increase it.

Implementation

Both verification and inference in a distributed multi-agent MSBN requires communication among agents. We designed a generic framework such that some modules can be reused and other modules only need to be modified slightly for different functions.

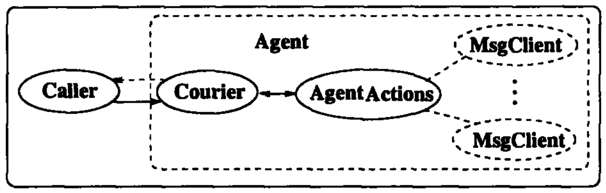

Figure 3: Each agent has a *Courier* and an *AgentActions*.

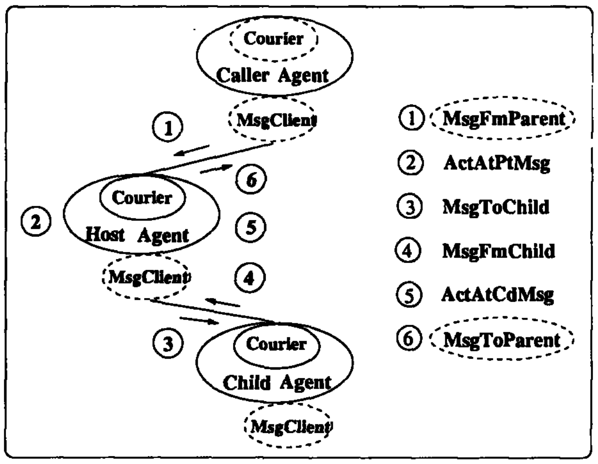

Figure 4: Each agent has six actions. Two (in dotted cycles) are performed by Courier, and the other by *AgentActions*.

Each agent has a *Courier* and an *AgentActions*, and can create multiple *MsgClients* when needed. *Courier* is responsible for receiving from caller and replying (if this agent is not Root) after the agent performs some actions according to the message type and the protocol. Courier is reused in all functions which involve communication. *AgentActions* perform four sequential actions after the Courier receives a message from the caller. If the agent is not a leaf in the hypertree, it creates multiple clients. Each client corresponds to one child agent and is called *MsgClient*, to communicate with one child agent. Figure 3 shows the structure.

Figure 4 shows the flow of a host agent's actions. Courier has two actions, *MsgFmParent* and *MsgToParent*. *MsgFmParent* performs an action which receives messages from the caller and processes the messages. *MsgToParent* performs an action to reply to

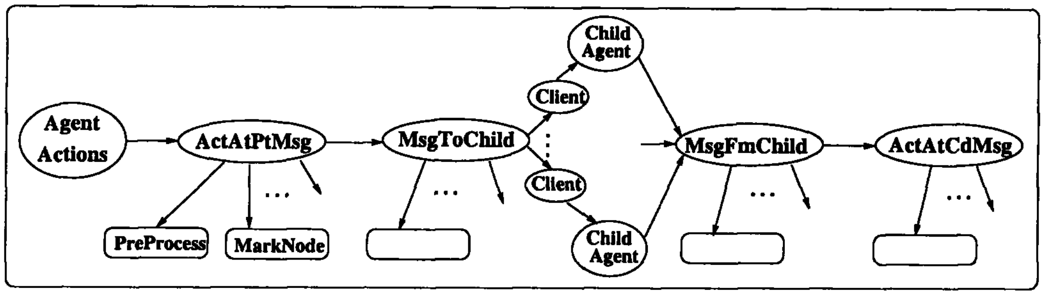

Figure 5: Four sequential actions of an agent. A leaf agent only has the first action.

the caller after the host agent performs some other actions. The four sequential actions of *AgentActions*, which are *ActAtPtMsg*, *MsgToChild*, *MsgFmChild* and *ActAtCdMsg*, are performed after the host agent receives a message from the caller (that is, after *MsgFmParent* is performed.). *ActAtPtMsg* is the action that an agent takes after it receives message from its caller. *MsgToChild* is an action which prepares messages for child agents and creates multiple clients to communicate with child agents. *MsgFmChild* is an action which waits for messages from all child agents and processes the messages it receives. The agent performs *ActAtCdMsg* after receiving reply from all child agents.

If an agent works as Root, it replies to no one and *MsgToParent* is not performed. If an agent is a leaf agent, it has no child agent and *MsgToChild* and *MsgFmChild* are not performed. Otherwise, an agent perform all six actions.

A special operation is performed in each action according to the type of the message it receives from its caller. For example, if the message type is "PREPROCESS", then operation of *ActAtParent* is to mark all the non-d-sepnodes which are leaf nodes or root nodes locally. The operation of *MsgToChild* is to set message type for each message and to create multiple MsgClients to communicate with all child agents. The operation of *MsgFmChild* is to wait for the reply from each child agent. Because *PreProcess* does not need to do anything after the agent receives reply from each child agent, the agent will not perform the last action-*ActAtChild*. Figure 5 shows an agent's sequential actions and the content of each action.

This framework can not only be used in the verification of MSBN, but also in other operations in a multiagent MSBN. For instance, we can implement triangulation or inference in a similar manner. *Courier*, *AgentActions MsgClient* can be used directly. What we need to change are the content in each action. *Inference* in distributed MSBN is thus implemented (Geng & Xiang 1998). Therefore, this design provides a unified framework for implementing all operations of MSBN including *Verification*, *Triangulation*, and *Inference*.

Conclusions

This paper discusses implementation of verification in distributed multi-agent MSBNs. We present a generic framework for implementing similar operations which requires communication among agents. We have tested the implementation experimentally. One limitation of the implementation is that when more than one agent try to initiate verification simultaneously, the system may not function correctly. Future research is needed to resolve the issue.

References

[1] Geng, H., and Xiang, Y. 1999. Implementation of Fully Distributed Inference in Multi-agent MSBN Systems. In *Proc. IEEE Canadian Conf. on Electrical and Computer Engineering*, 1698-1703, Edmonton.

[2] Jensen, F. V. 1996. *An Introduction to Bayesian Networks*, New York: Springer.

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

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

[5] Xiang, Y., Poole, D. and Beddoes, M.P. 1993. Multiply Sectioned Bayesian Networks and Junction forests for Large Knowledge-Based Systems. *Computational Intelligence.* 9(2): 171-220.

[6] Xiang, Y. 1996. A Probabilistic Framework for Cooperative Multi-agent Distributed Interpretation and Optimization of Communication. *Artificial Intelligence*, (87): 295-342.

[7] Xiang, Y. 1998. Verification of DAG Structures in Cooperative Belief Network Based Multi-agent Systems. *Networks*, 31: 183-191.