INFERENCE IN DISTRIBUTED MULTIAGENT REASONING SYSTEMS IN COOPERATION WITH ARTIFICIAL NEURAL NETWORKS

by

Abdunnaser Abdulhamid Diaf

A thesis submitted to the faculty of graduate studies
Lakehead University
in partial fulfillment of the requirements for the degree of
Masters of Science in Mathematical Science

Department of Computer Science

Lakehead University

March 2007

Your file Votre référence
ISBN: 978-0-494-31158-5
Our file Notre référence
ISBN: 978-0-494-31158-5

To my parents, who spent their lives, health and wealth, just as scented candles, for nothing but to see me and my siblings prosperous and healthy. To my wife for her continuous support, love, understanding and encouragement during my studies.

Acknowledgement

First of all, I greatly thank my supervisor Professor Nasser Noroozi and my co-supervisor Professor Francis Allaire. Mr. Noroozi has provided me invaluable guidance, continuous encouragement, support throughout this work. I am so grateful that he introduced me to the world of Artificial Intelligence. He has been very helpful and supportive both intellectually and personally. In all circumstances where I needed his approval for taking important decisions, he was supportive. While working with him, I always had the most genuine sense of freedom in pursuing research in my own way.

During my graduate studies, many Professors have influenced me significantly at Lakehead University: Professor Ruizhong Wei, Professor Maurice Benson and Professor Sabah Mohammed.

Also my appreciation to many professors and friends who have somehow been important during my residence in Canada.

Next, I would like to thank my brothers and sisters for their love and encouragement during my stay in Canada.

Contents

List of Tables

List of Figures

Abstract

This research is motivated by the need to support inference in intelligent decision support systems offered by multi-agent, distributed intelligent systems involving uncertainty. Probabilistic reasoning with graphical models, known as *Bayesian networks* (BN) or *belief networks*, has become an active field of research and practice in artificial intelligence, operations research, and statistics in the last two decades.

At present, a BN is used primarily as a stand-alone system. In case of a large problem scope, the large network slows down inference process and is difficult to review or revise. When the problem itself is distributed, domain knowledge and evidence has to be centralized and unified before a single BN can be created for the problem.

Alternatively, separate BNs describing related subdomains or different aspects of the same domain may be created, but it is difficult to combine them for problem solving, even if the interdependency relations are available. This issue has been investigated in several works, including most notably Multiply Sectioned BNs (MSBNs) by Xiang [Xiang93]. MSBNs provide a highly modular and efficient framework for uncertain reasoning in multi-agent distributed systems.

Inspired by the success of BNs under the centralized and single-agent paradigm, a MSBN representation formalism under the distributed and multi-agent paradigm has been developed. This framework allows the distributed representation of uncertain knowledge on a large and complex environment to be embedded in multiple cooperative agents and effective, exact, and distributed probabilistic inference.

What a Bayesian network is, how inference can be done in a Bayesian network under the single-agent paradigm, how multiple agents' diverse knowledge on a complex environment can be structured as a set of coherent probabilistic graphical models, how these models can be transformed into graphical structures that support message passing, and how message passing can be performed to accomplish tasks in model compilation and distributed inference are covered in details in this thesis.

The thesis is organized into six chapters. The first chapter, Chapter 1, introduces Bayesian networks as a concise representation of probabilistic knowledge and demonstrates the idea of belief updating by concise message passing using Pearl's algorithm (or $\lambda - \pi$ algorithm). Chapter 2 describes stepwise how to compile a BN

into a *junction tree* (JT) model and covers probabilistic inference by concise message passing under the single-agent paradigm. Also, algorithms for belief updating by passing potentials as messages in a JT are presented. Chapter 3 explains in details the framework for the distributed representation of probabilistic knowledge in a cooperative multi-agent reasoning system. It is shown that the MSBN restrictions follow directly from a set of very basic assumptions and required properties. It also presents a set of distributed algorithms used to compile an MSBN into a collection of related junction tree models, termed a linked junction forest (LJF) for effective multi-agent belief updating. A set of algorithms for performing effective, exact, and distributed inference by the agents in an MSBN organized as an LJF is covered in Chapter 4. In this chapter, we propose an efficient method that can guarantee performing belief assignment over all d-sepnodes in a hypertree. The method aims to relax the constraint of Xiang's model [Xiang93], namely the assumption of JPDs to be consistent with agent's belief. The model proposed can start from an arbitrary agent and its associated belief and establish the consistency in a practical manner. Chapter 5 addresses the issue of slow inference in multi-agent the model and how to overcome this problem by providing some enhancement to speed it up. A new model is introduced here using concepts from *artificial neural network* (ANN). An overview of ANN is given too. Chapter 6 presents an efficient object-oriented Bayesian network framework built by C++. The package deals with a BN entities as objects starting with a simple variable and ending with an agent (a hyper node). Implementation results for a simple BN, a single-agent JT, and a multi-agent hyper tree are shown as well (including the compilation processes for both, JT and LJF models).

Chapter 1

Introduction: Bayesian Network Theory

A *Bayesian network* (BN) is a form of probabilistic graphical model, named after Thomas Bayes (1702-1761). A Bayesian network, also known as *Bayesian belief network*, *belief network*, *probabilistic network*, *causal network*, or *influence diagram*, is a graphical structure for representing the probabilistic relationships among a number of variables and doing probabilistic inference with those variables [17].

1.1 Bayes' Theorem

Probability theory has adopted the autoepistemic phrase "...given that Y is known" as a primitive of the language [18]. Syntactically, this is denoted by placing Y behind the conditioning bar, "|", in a statement such as $P(X|Y) = p$. This statement combines the notions of knowledge and belief by attributing to X a degree of belief p, given the knowledge Y. Y is also called the *context* of the belief in X, and the notion $P(X|Y)$ is called *Bayes conditionalization*. Thomas Bayes made his main contribution to the science of probability by associating the English phrase "...given that I know Y" with the now-famous ratio formula:

$$P(X|Y) = \frac{P(X,\ Y)}{P(Y)} \tag{1.1}$$

Eq. 1.1 has become a definition of *conditional probabilities*. The probability of an event X conditional on another event Y, $P(X|Y)$, is generally different from the probability of Y conditional on X, $P(Y|X)$. However, there is a definite relationship between the two, and *Bayes's theorem* is the statement of that relationship [1].

1.1.1 Bayes' Theorem

Given two events X and Y such that $P(X) \geq 0$ and $P(Y) \geq 0$, then the following equality follows directly from Eq. 1.1 [16]:

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} \tag{1.2}$$

where:

- *P(X| Y)* is the posterior probability of the hypothesis X, given the data Y,
- *P(Y|X)* is the probability of the data Y, given the hypothesis X, or the likelihood of the data,
- *P(X)* is the prior probability of the hypothesis X, and
- *P(Y)* is the prior probability of the data Y, or the evidence.

Furthermore, given n mutually exclusive and exhaustive events $X_1, X_2, ...X_n$ such that $P(X_i) \geq 0$ for all i_s, we have for $1 \leq i \leq n$,

$$P(X_j|Y) = \frac{P(Y|X_j)P(X_j)}{\sum_{i=1}^{n} P(Y|X_i)P(X_i)} \tag{1.3}$$

Two other rules are important in Bayesian networks, the *expansion rule* and the *chain rule.*

1.1.2 Expansion rule

Consider the situation where X and Y are random variables with n possible outcomes:

$$\begin{aligned} P(X) &= P(X|y_1).P(y_1) + P(X|y_2).P(y_2) + \ldots + P(X|y_n).P(y_n) \\ &= \sum_{i=1}^{n} P(X|y_i).P(y_i) \quad or \quad = \sum_{Y} P(X|Y).P(Y) \end{aligned} \tag{1.4}$$

Applying the expansion rule means that variables can be introduced on the right-hand side of the equation, as long as we sum over all their possible values. This concept is also known as the *marginal probability* of X, meaning that only the probability of one variable, X, is important and all information about other variables, Y, is ignored.

1.1.3 Chain rule

This rule is derived by writing Bayes' Theorem in the following form, which is called the *product rule*:

$$\begin{aligned} P(X,Y) &= P(Y|X).P(Y) \\ &= P(X|Y).P(X) \end{aligned} \tag{1.5}$$

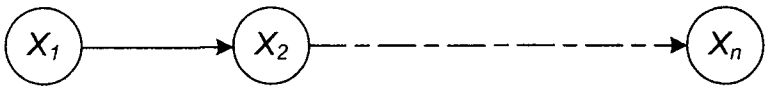


Figure 1.1: A chain DAG over a universe $U = (X_1, X_2, ..., X_n)$

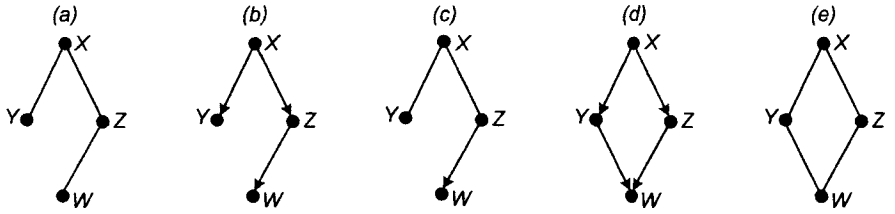


Figure 1.2: (a) An undirected graph; (b) A directed graph; (c) A hybrid graph; (d) A multiply connected directed graph; (e) A multiply connected undirected graph.

Considering the simple Bayesian network shown in figure 1.1, let $U = (X_1, X_2, ..., X_n)$ be a universe of variables. Successive application of the product rule in the following, yields the chain rule:

$$\begin{aligned} P(U) &= P(X_1, X_2, ..., X_n) \\ &= P(X_1, ..., X_{n-1}).P(X_n|X_1, ..., X_{n-1}) \\ &= P(X_1, ..., X_{n-2}).P(X_{n-1}|X_1, ..., X_{n-2}).P(X_n|X_1, ..., X_{n-1}) \\ &\vdots \\ &= P(X_1).P(X_2|X_1)...P(X_n|X_1, ..., X_{n-1}) \\ &= P(X_1).\prod_{i=2}^{n}(X_i|X_1, ..., X_{i-1}) \end{aligned} \tag{1.6}$$

Therefore, the joint probability distribution $P(U)$ is the product of all conditional probabilities specified in a Bayesian network:

$$P(U) = \prod_{i=1}^{n}(X_i|\pi(X_i)) \tag{1.7}$$

where $\pi(X_i)$ is the parent set of X_i.

1.2 Basic Definitions

1.2.1 Graph (G)

A graph is a symbolic representation of a network and of its connectivity. It models an abstraction of the reality into a set of linked nodes. A graph G is a set of vertices (*nodes*) V connected by edges (*links*) E. Thus $G = (V, E)$. Graphs

can be of different types such as *directed* (where all links in the graph have directions, figure 1.2(b)), *undirected* (where all links have no directions, figure 1.2(a)), and *hybrid* (where the graph has both kinds of links, see figure 1.2(c)) [21]. For *undirected* graphs, terminologies are introduced to describe the neighborhood of a node, a path from one node to another, a cycle in a graph, a graph within another graph, and other aspects of connectivity of a graph. For *directed* graphs, additional terminologies are introduced to describe a directed path or cycle and to name nodes at different locations on a directed path.

Vertex (Node)

A node v is a terminal point or an intersection point of a graph. We can use nodes to represent events. Each node contains a *conditional probabilistic table, CPT*, that contains probabilities of the node being a specific value given the values of its parents.

Edge (Link)

An edge e is a link between two nodes. The link (X, Y) between the pair of nodes X and Y shows a direct relationship between them. A link is called *directed* or *unidirectional* when it is represented as an arrow and may be called an *arc*. When an arrow is not used, it is assumed the link is *bidirectional*. A unidirectional arc (X, Y) is directed from X (called the tail of the arc) to Y (called the head of the arc). We also refer to X as a *parent* of Y and to Y as a *child* of X.

1.2.2 Path ρ

A path ρ in a directed graph is a directed path if each node in ρ, other than the first and the last, is the head of one arc in and the tail of the other arc in ρ. If there is a directed path from a node X to a node Y $\langle X, N_1, N_2, \ldots, N_k, Y\rangle$, then X is called an ancestor of Y and Y is called a descendant of X. A path is a cycle if it contains two or more distinct nodes and the first node is identical to the last node.

A graph is *connected* if there exists a path between every pair of nodes, as all graphs in Figure 1.2). A connected graph is a *tree* if there exists exactly one path between every pair of nodes, as graphs in (a), (b), and (c); otherwise, it is *multiply connected* as graphs in (d) and (e).

1.2.3 Directed Acyclic Graph (DAG)

A directed graph G is acyclic or is a directed acyclic graph if it contains no directed cycles. A directed acyclic graph is often referred to as a DAG. In Figure 1.2, the directed graphs in (b) and (d) are DAGs. The graph in (b) is singly connected and has no cycles. The graph in (d) is multiply connected and while $\langle X, Y, W, Z, X\rangle$ is a cycle, it is not a directed cycle.

1.2.4 D-Separation

In a DAG, *d-separation* is a property of two nodes x and y with respect to a set of nodes Z. x and y are said to be *d-separated* by Z if no information can flow between them when Z is observed [9]. The d in d-separation stands for "directed acyclic graph". Informally, two variables x and y are independent conditional on Z if knowledge about x gives you no extra information about y once you have knowledge of Z. In other words, once you know Z, x adds nothing to what you know about y. Formally, a path ρ between two variables x and y is *closed* or *blocked* by a set of nodes Z if *(1) There exists $z \in Z$ that is either a fork (tail-to-tail) or a chain (head-to-tail) on ρ or (2) There exists a node v that is a collider (head-to-head) on ρ and neither v nor any descendant of v is in Z.* If both conditions *fail*, then ρ is rendered *open* by Z. Nodes x and y are d-separated by Z if every path between x and y is *closed* by Z;

In a simple chain, $X \longrightarrow Z \longrightarrow Y$, if X is d-separated from Y conditional on Z, then X is independent of Y conditional on Z. The following formula expresses the result.

$$P(X,Y|Z) = P(X|Z)P(Y|Z) \tag{1.8}$$

1.3 Bayesian Networks

A Bayesian network represents and processes probabilistic knowledge. The representational components of a BN are a *qualitative* and a *quantitative* component.

The *qualitative* component encodes a set of conditional dependence and independence statements among a set of random variables, informational precedence, and preference relations. The statements of conditional dependence and independence, information precedence, and preference relations are visually encoded using a graphical structure.

The *quantitative* component, on the other hand, specifies the strengths of dependence relations expressed by the assignment of the joint probability distributions (JPDs) to the nodes.

The graphical representation of a Bayesian network describes knowledge of a problem domain in a precise manner. It is intuitive and easy to comprehend, making it an ideal tool for communication of domain knowledge between experts, users, and systems [14]. For these reasons, the formalism of probabilistic networks is becoming an increasingly popular domain knowledge representation for reasoning and decision making under uncertainty.

Furthermore, a Bayesian network is represented as a directed graphical model whose nodes signify events (variables), and unidirectional links represent causal dependence relations among the variables. In addition, there is a local probability distribution for each variable given the values of its parents. Nodes can represent

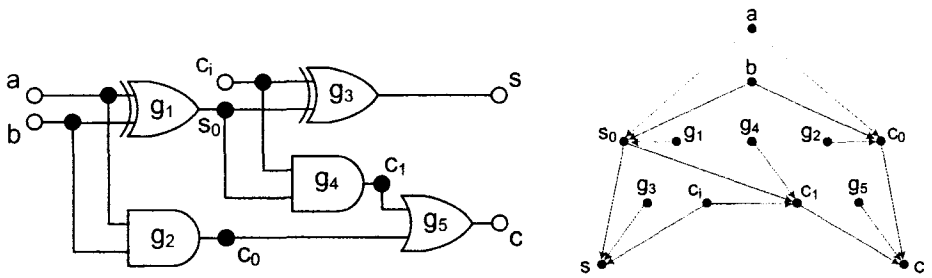


Figure 1.3: The graphical model for the digital circuit.

any kind of variables, be it a measured parameter, a latent variable or a hypothesis. They are not restricted to representing random variables. If there is an arc from node X to another node Y, then variable Y depends directly on variable X, and X is called a parent of Y. Figure 1.3 shows the graph of a Bayesian network representing knowledge of a full-adder digital circuit.

A Bayesian network can also be defined as a framework for explaining causal relations consisting of a set of variables (each with a finite set of mutually exclusive states) connected by a set of directed edges. Probability calculus is used to describe the probabilistic relationship of each variable with its parents.

1.4 Why a Bayesian network?

There are a number of models available for data analysis, including rule bases, decision trees and artificial neural networks. There are also several techniques for data analysis such as classification, density estimation, regression and clustering. One may wonder what Bayesian networks and Bayesian methods have to offer to solve such problems and why Bayesian networks are so interesting. The following paragraphs provide five answers to the question [7].

First, Bayesian networks can handle incomplete data sets without difficulty because they discover dependencies among all variables. When one of the inputs is not observed, most models will end up with an inaccurate prediction. That is because they do not incorporate a correlation between the input variables. Bayesian networks suggest a natural way to encode these dependencies.

Second, Bayesian networks allow one to learn about causal relationships between variables. There are two important reasons to learn about causal relationships. The process is worthwhile when we would like to understand the problem domain, for instance, during exploratory data analysis or when an agent is exploring the environment. Additionally, in the presence of intervention, one can make predictions with the knowledge of causal relationships.

Third, considering the Bayesian statistical techniques, it is possible to combine expert knowledge and data into a BN. Prior or domain knowledge is crucially im-

portant if one performs a real-world analysis; in particular, when data is inadequate or expensive. The encoding of causal prior knowledge is straightforward because Bayesian networks have causal semantics. Additionally, Bayesian networks encode the strength of causal relationships with probabilities. Therefore, prior knowledge and data can be put together with well-studied techniques from Bayesian statistics.

Fourth, Bayesian methods give an efficient approach to avoid the over-fitting of data during learning. Models can be smoothed in such a way that all available data can be used for training by using Bayesian approach.

Finally, Bayesian networks are able to model causal relationships between variables.

1.4.1 Advantages of using Bayesian networks

For over a decade, expert systems have used BNs in domains where uncertainty plays an important role. Nowadays, BNs appear in a wide range of diagnostic medical systems, fraud detection systems, missile detection systems, decision support systems, engineering, text analysis, image processing, data fusion, and spam filtering [4].

In addition, Bayesian networks have a couple of properties that make them popular and suitable. Eleven important properties are, in no particular order, the following:

- BNs model probabilities naturally,
- BNs can derive posterior beliefs given some observations,
- BNs have a bidirectional message passing architecture,
- BNs have the ability to imitate human thinking processes,
- BN can be used as a classifiers,
- BNs can explain non obvious causal relationships,
- BNs can find the variables with the most impact,
- BNs can learn details about their own structure from raw data,
- BNs can model expert (subjective) knowledge,
- several useful algorithms exist, and
- BNs are simple and easy to learn.

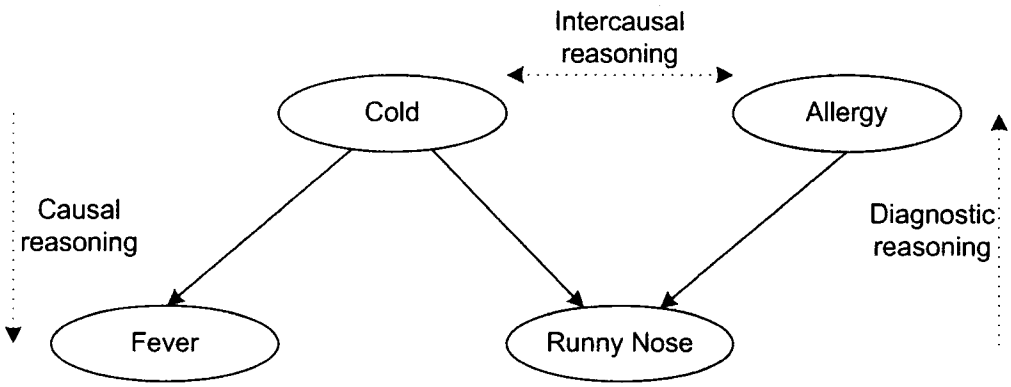


Figure 1.4: BNs support three kinds of reasoning: causal, diagnostic, and intercausal.

1.5 Bayesian inference

Inference is a process whose main goal is to calculate the probability function $P(Query|Evidence)$. The evidence is expressed as an instantiation of some variables in the BN. This section gives the details of an algorithm for exact inference in Bayesian networks, namely *Pearl's message-passing algorithm.* Before we present Pearl's, let us introduce what kinds of inference Bayesian networks can support.

1.5.1 Types of inference in BNs

BNs can be seen as compact representations of "fuzzy" cause-effect rules that are capable of performing *causal* and *diagnostic* inference as well as *intercausal* inference (see Figure 1.4) [10].

Causal inference

Causal inference (top-down, $P(Effect|Cause)$) follows the direction of the causal links between variables of a model; e.g., $P(Fever|Cold = true)$ or $P(RunnyNose|Cold = true)$.

Diagnostic inference

Diagnostic inference (bottom-up, $P(Cause|Effect)$) goes against the direction of the causal links; e.g., $P(Cold|RunnyNose = true)$ and $P(Allergy|RunnyNose = true)$ provide either cold or allergy being the correct diagnosis. Diagnostic inference is a common task in expert systems.

Intercausal inference

Intercausal inference, or *explaining away*, $(P(Cause|Cause))$ in a Bayesian network occurs when evidence that establishes a cause for an event reduces the likelihood

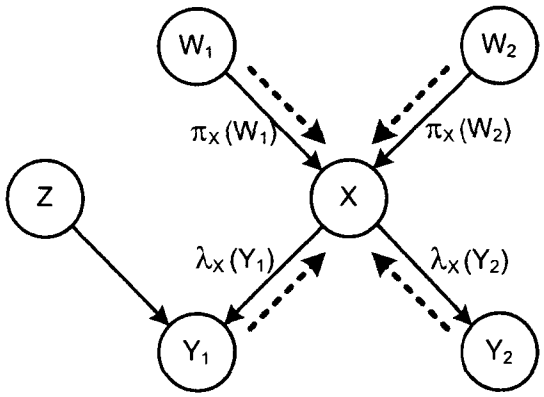


Figure 1.5: Pearl's *(or $\lambda - \pi$)* message-passing algorithm.

of other possible causes [5].

1.5.2 Pearl's (or $\lambda - \pi$) message-passing algorithm

This algorithm depends on massage passing among nodes in the DAG to perform exact inference. Given a set e of values of a set E of instantiated variables, the algorithm determines $P(x|e)$ for all values x of each variable X in the network. It accomplishes this by generating messages from each instantiated variable to its neighbors. The neighbors in turn, propagate messages to their neighbors.

Pearl's message-passing algorithn, also known as $\lambda - \pi$ *algorithm*, is applicable for three topologies of BNs, *chain*, *rooted-tree*, and *singly-connected* DAGs. Inference in the forth topology, *multiply-connected* DAGs, can also be accomplished by using this algorithm but in an indirect manner or method called *loop cutset conditioning* [18]. Conditioning is the ability to form a multiply-connected DAG from a number of singly-connected DAGs by instantiating a selected group of variables.

Since all DAG topologies are formed as singly-connected DAGs, we present only inference in singly-connected DAGs.

Inference in singly-connected DAGs

The two rules that are needed for inference are Bayes' theorem (Eq. 1.2) and the expansion rule (Eq. 1.4).

Let us consider that we want to know the probability of each value of the variable X in Figure 1.5 given that evidence might come through its parents and children. *Let Y_X be the subset of E containing all members of E that are in the subtree connected with X through its children, and let Z_X be the subset of E containing all members of E that are in the subtree connected with X through its parents. We have for each value of x,*

$$
\begin{aligned}
P(x|e) &= P(x|e_{D_X}, e_{A_X}) \\
&= \frac{P(e_{D_X}, e_{A_X}|x)P(x)}{P(e_{D_X}, e_{A_X})} \Leftarrow \textit{Baye's Theorem, Eq.}(1.2) \\
&= \frac{P(e_{D_X}|x)P(e_{A_X}|x)P(x)}{P(e_{D_X}, e_{A_X})} \Leftarrow d-\textit{separation Eq.}(1.8) \\
&= \frac{P(e_{D_X}|x)P(x|e_{A_X})P(e_{A_X})P(x)}{P(x)P(e_{D_X}, e_{A_X})} \Leftarrow \textit{Baye's Theorem, Eq.}(1.2) \\
&= \frac{P(e_{A_X})}{P(e_{D_X}, e_{A_X})} P(e_{D_X}|x)P(x|e_{A_X}) \\
\textit{Letting} \quad & \alpha = \frac{P(e_{A_X})}{P(e_{D_X}, e_{A_X})}, \ \lambda(x) = P(e_{D_X}|x), \ \pi(x) = P(x|e_{A_X}), \ \textit{then} \\
P(x|e) &= \alpha\lambda(x)\pi(x) \qquad (1.9)
\end{aligned}
$$

α is a constant that does not depend on the value of x, $\lambda(x)$ is called *"$\lambda-value$"* of x and depends on $\lambda-messages$ coming from its children, and $\pi(x)$ represents *"$\pi-value$"* of x and depends on $\pi-messages$ coming from its parents.

The following equations and equalities describe how $\lambda-value$ and $\pi-value$ can be obtained, and what $\lambda-messages$ and $\pi-messages$ represent.

1. λ ***value:***

   If $X \in E$ and X's value is x^*, then

$$\lambda(x^*) \equiv 1, \ \lambda(x : x \neq x^*) \equiv 0$$

   If $X \notin E$ and X is a leaf, for all values of x,

$$\lambda(x) \equiv 1$$

   If $X \notin E$ and X is a notleaf, for all values of x,

$$
\begin{aligned}
\lambda(x) &= P(e_{D_X}|x) \\
&= P(e_{Y_1}, e_{Y_2}|x) \Leftarrow e_{D_X} : \{e_{Y_1}, e_{Y_2}\} \\
&= P(e_{Y_1}|x)P(e_{Y_2}|x) \Leftarrow D-\textit{Separation Eq.}(1.8) \\
&= \prod_{i=1}^{n} \lambda_{Y_i}(x) \Leftarrow \lambda_{Y_i}(x) = P(e_{Y_i}|x) \qquad (1.10)
\end{aligned}
$$

2. π ***value:***

   If $X \in E$ and X's value is x^*, then

$$\pi(x^*) \equiv 1,\ \pi(x : x \neq x^*) \equiv 0$$

If $X \notin E$ and X is a root, for all values of x,

$$\pi(x) \equiv P(x)$$

If $X \notin E$ and X is a notroot, for all values of x,

$$\begin{aligned}
\pi(x) &= P(x|e_{A_X}) \\
&= P(x|e_{W_1}, e_{W_2}) \Leftarrow e_{A_X} : \{e_{W_1}, e_{W_2}\} \\
&= \sum_{w_1,w_2} P(x|w_1,w_2)P(w_1,w_2|e_{W_1},e_{W_2}) \Leftarrow Eq.(1.4) \\
&= \sum_{w_1,w_2} P(x|w_1,w_2)P(w_1|e_{W_1})P(w_2|e_{W_2}) \Leftarrow Eq.(1.8) \\
&\quad \pi_x(w_i) = P(w_i|e_{W_i}),\ then \\
&= \sum_{w_1,w_2,\ldots,w_m} \Big(P(x|w_1,w_2,\ldots,w_m)\prod_{i=1}^{m}\pi_x(w_i)\Big). \qquad (1.11)
\end{aligned}$$

3. λ ***message:***

   For each child X of W_i, for all values of w_i,

$$\lambda_X(w_i) \equiv \sum_x \lambda(x) \sum_{w_j : j\neq i} \Big(P(x|w_1,\ldots,w_m)\prod_{j\neq i}\pi_X(w_j)\Big). \qquad (1.12)$$

   where $W_{j,j\neq i}$ are the other parents of X.

4. π ***message:***

   Let X be a parent of Y_i. Then for all values of x,

$$\pi_{Y_i}(x) \equiv \prod_{j\neq i} \lambda_{Y_j}\pi(x) \qquad (1.13)$$

The algorithm:

All formulas involved in inference in singly-connected DAGs just have been mentioned with details. We shall now summarize the steps of the algorithm considering a typical node X having m parents, $W_1, W_2, \ldots, W_m$, and n children, $Y_1, Y_2, \ldots, Y_n$, as in Figure 1.5.

Three types of parameters need to be available in order to compute the belief distribution of variable X:

1. The current strength of the *causal* support $\pi - message$ contributed by each incoming link $W_i \rightarrow X$ (from Eq. 1.13).

2. The current strength of the *diagnostic* support $\lambda - message$ contributed by each outgoing link $X \rightarrow Y_j$ (from Eq. 1.12).

3. The CPT of X with respect to its immediate parents, $P(x|w_1, w_2, ..., w_m)$.

Using these parameters ready, local belief updating can be performed in three steps:

1. **Belief updating:** When node X is activated, it simultaneously inspects the messages $\pi_X(W_i)$, $i = 1, ..., m$ communicated by its parents and the messages $\lambda_{Y_j}(x)$, $j = 1, ..., n$ communicated by its children. Using these inputs, it updates its belief measure according to Equations 1.10, 1.11 and 1.9. α is a normalizing constant rendering $\sum_x P(x) = 1$.

2. **Diagnostic message-passing (Bottom-up):** Using the messages received, node X, according to Equation 1.12, computes new $\lambda - messages$, $\big(\lambda_X(w_1)$, $\lambda_X(w_2)$, ..., $\lambda_X(w_m)\big)$, to be sent to its parents.

3. **Causal message-passing (Top-down):** Each node, according to Equation 1.13, computes new $\pi - messages$, $\big(\pi_{Y_1}(x)$, $\pi_{Y_2}(x)$, ..., $\pi_{Y_n}(x)\big)$, to be sent to its children.

Chapter 2

From *Directed Acyclic Graphs* to *Junction Trees*

The junction tree propagation method (Jensen [11]) involves the extraction of an undirected graph from a DAG in the Bayesian network, and the creation of a tree whose vertices are the cliques of a triangulated graph [12].

2.1 The Junction Tree framework

The method is a *message-passing* algorithm, where *messages* are seen as local computations involving two maximal cliques of the graph. Its general view for undirected graphical models can be described as follows:

1. Moralization. The first step is to generate a moral graph by connecting the parents of each node pairwise and making all edges in the graph undirected.
2. Triangulation. This step is to triangulate the moral graph, or equivalently to make it a chordal graph by the proper insertion of new edges.
3. Junction Tree construction. A hyper-graph, whose nodes are the maximal cliques of the chordal graph, is constructed.
4. Potentials assignment. The nodes of the Junction Tree are initialized with a product of probability distributions in a proper manner.
5. Propagation. A *message-passing* or *potentials propagation* algorithm is run on the Junction Tree in order to systematically update the potentials and bring it into consistency.

2.1.1 The Junction Tree definition

Trees of cliques are a concept from graph theory that play an important role in problems like probabilistic inference, constraint satisfaction, query optimization,

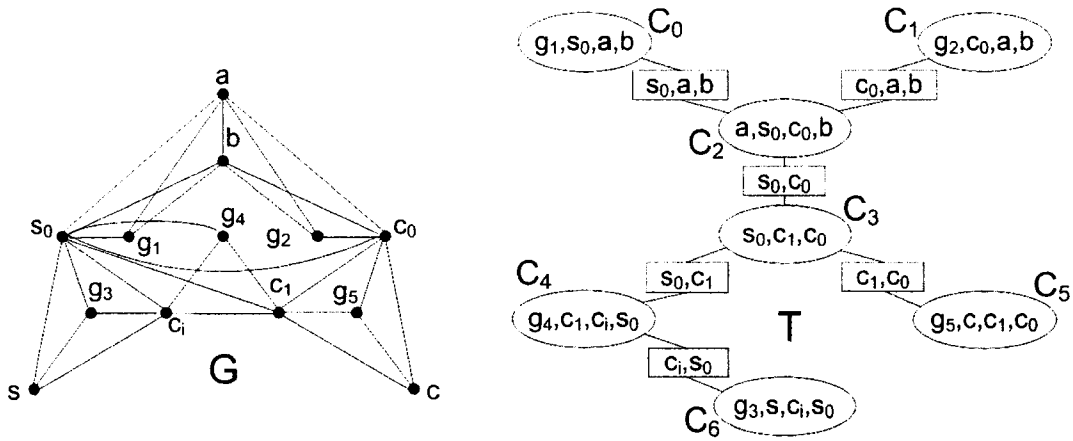


Figure 2.1: T is the *junction tree* of the undirected graph G.

and matrix decomposition. Clique trees are also called *junction trees* or *join trees.*

Beliefs in the original Bayesian network are updated by passing messages among the nodes as we have seen in Section 1.5 on page 8. Once the junction tree is built, we no longer in need of the original DAG.

Definition

A *junction tree (JT)* is a tree T whose nodes and edges are annotated with sets of vertices from an undirected graph G. The vertex set associated with a junction tree node is called the node's clique. The vertex set associated with a junction tree edge is called the edge's separator, and it is defined to be the intersection of the two incident nodes' cliques.

A valid junction tree T for an undirected graph G should satisfy two properties:

- Every clique of G is contained in at least one clique of T; and
- the cliques of T satisfy the *running intersection property*: For each pair U, V of cliques in the junction tree T with intersection S, all cliques on the unique path between them contain S.

An example of an undirected graph G and its junction tree T are shown in Figure 2.1.

In order to construct a junction tree from a Bayesian network. A series of graphical transformations that result in a joint tree can be summarized as follows:

1. Construct an undirected graph, called a *moral graph*, from the directed acyclic graph (DAG) representation of the Bayesian network.
2. Triangulate the moral graph by adding arcs selectively to form a *chordal graph.*

3. From the chordal graph, identify selected subsets of nodes, called cliques.

4. Build a junction tree: connect the cliques to form an undirected tree that satisfies junction tree properties, and insert appropriate separators.

2.2 Moral Graph

A moral graph is an intermediate undirected graphical representation to facilitate the conversion of a DAG model to a junction tree model.

Given the DAG G of a Bayesian network. For each child node in G, connect its parent nodes pairwise and make all links in the graph undirected. The resultant undirected graph G' is the ***moral graph*** *of G.*

For example, the undirected graph in Figure 2.2(b) represents the moral graph of the DAG in (a).

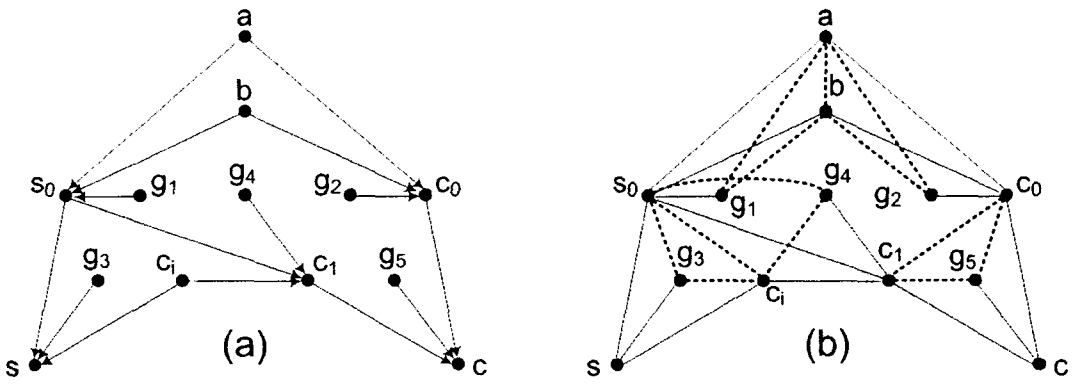


Figure 2.2: From DAG (a) to undirected moral graph (b)

2.3 Graph triangulation (Chordal graph)

A JT can be derived from the triangulated moral graph of the DAG rather than from the DAG directly. This is because a JT uniquely defines an undirected graphical structure, and hence a trangulated moral graph of the DAG provides a more direct basis to work on than the DAG itself. For example, consider the JT T in Figure 2.1. Construct an undirected graph G' with the generating set of the JT as the nodes. For each pair of nodes contained in a cluster in the JT, connect the pair in G'. The resultant G' is the graph G in Figure 2.1.

To build a JT, the cliques must be determined. In a JT, each cluster admits no graphical separation and signifies no conditional independence internally. Similarly, in an undirected graph G, a set of pairwise-connected nodes admits no graphical separation.

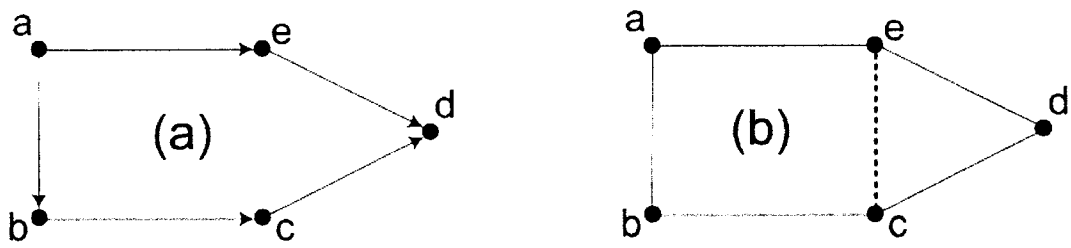


Figure 2.3: (a) A DAG. (b) The undirected moral graph of (a)

A set of nodes in an undirected graph G is ***complete*** *if they are pairwise connected. A maximal set of nodes that is complete is called a* ***maximal clique*** *in G.* Let $\Omega(G)$ be the set of maximal cliques of G.
Because a clique is a maximal set of variables without graphically identifiable conditional independence, it should become a *cluster* in the JT representation. The JT T in Figure 2.1 has C_0 through C_6 as its clusters.

Unfortunately we can not construct a JT from cliques of every moral graph G. *Given a connected undirected graph G and* $\Omega(G)$*, then there exists a junction tree T whose clusters are elements of* $\Omega(G)$ *if and only if G is* ***chordal*** [21].
As an example, consider the DAG in Figure 2.3 (a) with its moral graph in (b), the cliques are $\{a, b\}$, $\{a, e\}$, $\{b, c\}$, $\{c, d, e\}$. No cluster graph made out of these clusters is a JT.

Given an undirected graph G, a path or cycle ρ has a chord if there is a link in G between two nonadjacent nodes on ρ. G is *chordal* or *triangulated* if every cycle of length greater than 3 contains a chord. A cycle of length > 3 without a chord is called a *chordless cycle.* The graph G in Figure 2.1 is chordal, but in Figure 2.3(b) is not because the cycle $\langle a, b, c, e, a\rangle$ of length 4 does not have a chord.

A node v in an undirected graph G is *simplicial* if the nodes adjacent to it, $adj(v)$, are complete. Figure 2.3(b) has one simplicial node d. A chordal graph G has at least one simplicial node [21].

Definition

A *Chordal graph* establishes a relation between *moral graphs* and *junction trees.* It is shown that a moral graph model must be converted to a chordal graph in order to construct a junction tree model.
The next subsection presents an algorithm known as *triangulation by elimination* for converting a moral graph into a chordal graph. Elimination also provides a simple way to check if a graph is chordal.

2.3.1 Converting the moral graph into a chordal graph

A simple operation called *node elimination* is introduced here so it can be used for triangulating a moral graph.

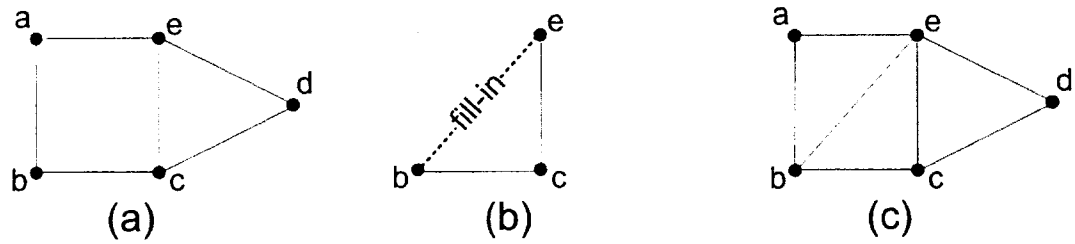


Figure 2.4: Elimination of node a with one fill-in. Node d is already simplicial.

Given an undirected moral graph G. A node v is eliminated from G if and only if v is simplicial. Otherwise, links are added to G to make v simplicial and then v is eliminated. The necessary added links are called *fill-ins*. Figure 2.4 illustrates the elimination of nodes d and a. Because d is already simplicial, no fill-ins were needed. Elimination of node a required the fill-in $\langle b, e \rangle$. Each of b, c and e can be subsequently eliminated in an order, because they are now simplicial. No more fill-ins need to be added.

An undirected moral graph G is *eliminatable* if all nodes can be eliminated in some sequence without any fill-ins. Nodes in Figure 2.4(c) can be eliminated in the order (d, a, b, c, e) without fill-ins; hence, the graph is eliminatable. Note that if the graph is eliminated in the order (d, b, a, c, e), a fill-in $\langle a, c \rangle$ needs to be added when eliminating b. Consequently, as long as there exists one elimination order that is fill-in free, the graph is eliminatable. On the other hand, if no such order can be found, the graph is not eliminatable.

In fact, an undirected moral graph G is chordal if and only if it is eliminatable [21].

Triangulation by elimination algorithm [21, 23] provides a straightforward way to check if a graph is chordal.

Algorithm (IsChordal)

Given an undirected moral graph $G = (V, E)$, do the following lines and return $TRUE$ if G is chordal; $FALSE$ otherwise.

for $i = 1$ *to* $|V|$, *do*
  search for a simplicial node v;
  if found, eliminate v;
  else return $false$;
return true;

If G is not chordal, elimination of nodes will require fill-ins. If these fill-ins are added back to G, the resultant graph G' will be chordal.

Given an undirected moral graph $G = (V, E)$ and a set of fill-ins F produced by eliminating all nodes of G in any order, then $G' = (V, E \cup F)$ is eliminatable.

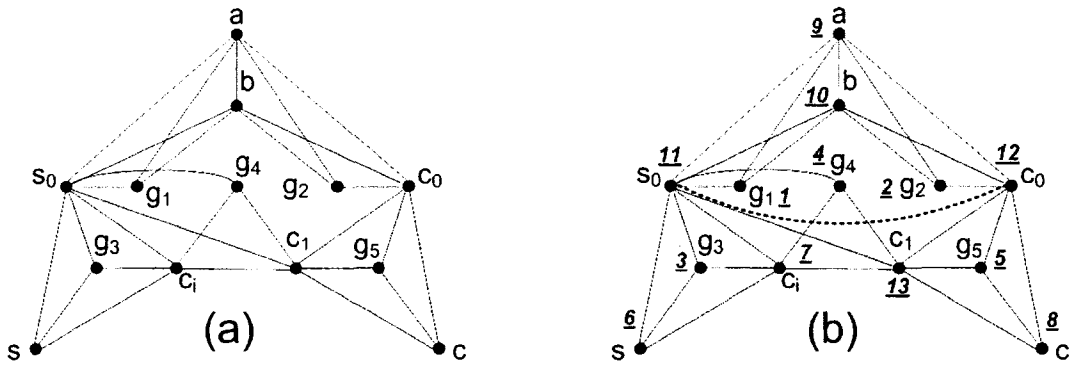


Figure 2.5: (b) The resultant chordal graph after applying *TriangulationByElimination* algorithm on the undirected graph in (a).

In general, we can obtain an eliminatable graph from an undirected graph by making a simple modification of algorithm IsChordal: At each iteration of the *for* loop, if a simplicial node cannot be found, eliminate a node with the minimum number of fill-ins needed and store the fill-ins. At the end of the for loop, return $G' = (V, E \cup F)$.

Algorithm (GetChordalGraph)

Given an undirected moral graph $G = (V, E)$ do the following lines and return a chordal graph $G' = (V, E')$ where $E' = E \cup F$

$F = \emptyset$
for $i = 1$ *to* $|V|$, *do*
  search in G for simplicial node v;
  if found, eliminate v;
  else
    select a node w with the minimum number of fill-ins required to eliminate w;
    add fill-ins produced to F;
    add fill-ins to G;
    eliminate w;
return $G' = (V, E \cup F)$;

Figure 2.5 illustrates how the chordal graph can be obtained from a moral graph by using the above algorithm, *triangulation by elimination.* Underlined numbers show the elimination sequence and dashed lines represent the fill-ins links.

The triangulation process has been introduced in order to convert the moral graph of a BN into a chordal graph so that a JT model can be built. Fill-ins added during triangulation causes the loss of some graphical separation relations and hence should be kept minimal. Unfortunately, the minimality in selecting w of

algorithm *GetChordalGraph* does not guarantee finding the chordal graph with the minimum number of fill-ins. The problem is NP-complete [21].

2.4 Constructing Junction Trees from Chordal Graphs

For a fixed moral graph, the elimination sequence determines the triangulated graph. The maximal cliques of this triangulated graph determine a junction graph and from that: a junction tree.

In this section, we assume that G is a triangulated or chordal graph.

In order to construct a JT T from the triangulated moral graph G of a BN, the set $\Omega(G)$ need to be identified. Once $\Omega(G)$ is determined, a junction graph $H = (V, \Omega(G), E)$ is defined. Then, by removing specific separators from E on H, a junction tree T can be obtained.

2.4.1 Identifying Cliques

Given a chordal graph G and its elimination sequence $\gamma = (v_1, v_2, ..., v_n)$, if each node v_i and its adjacenct nodes are saved as a clique $C_i = v_i \cup adj(v_i)$ just before it is eliminated, the resulting record will contain all the possible maximal cliques of G. In order to obtain $\Omega(G)$, every clique C_i that is contained in another clique C_j needs to be removed.

2.4.2 From $\Omega(G)$ to an efficient JT representation

Given a graph of cliques $H = (V, \Omega, E)$, a junction tree T can be constructed by forming a maximal spanning tree from H. A tree of cliques will be constructed with separators. We have to mention here that not every clique tree is a junction tree. *A clique tree is a junction tree if and only if it satisfies the running intersection property.* In particular, a junction tree is a maximum weight spanning tree of H where the weights on the separators are the number of variables shared by the two cliques [12].

Figure 2.6 shows that not all the clique trees obtained from a set Ω of cliques, generated from a chordal graph G, are junction trees.

Once the set Ω obtained, an algorithm called *greedy algorithm, or Prim's algorithm for a maximal spanning tree*, will be used to identify how Ω members will be connected and then what the set E of links will be.

Greedy Spanning Tree Algorithm

In general, for maximal spanning tree problems on a graph of vertices, each link is assigned a weight. The goal of the problem is to select a spanning tree so that the total of the weights of the selected links is maximized [6]. For a graph of cliques, the

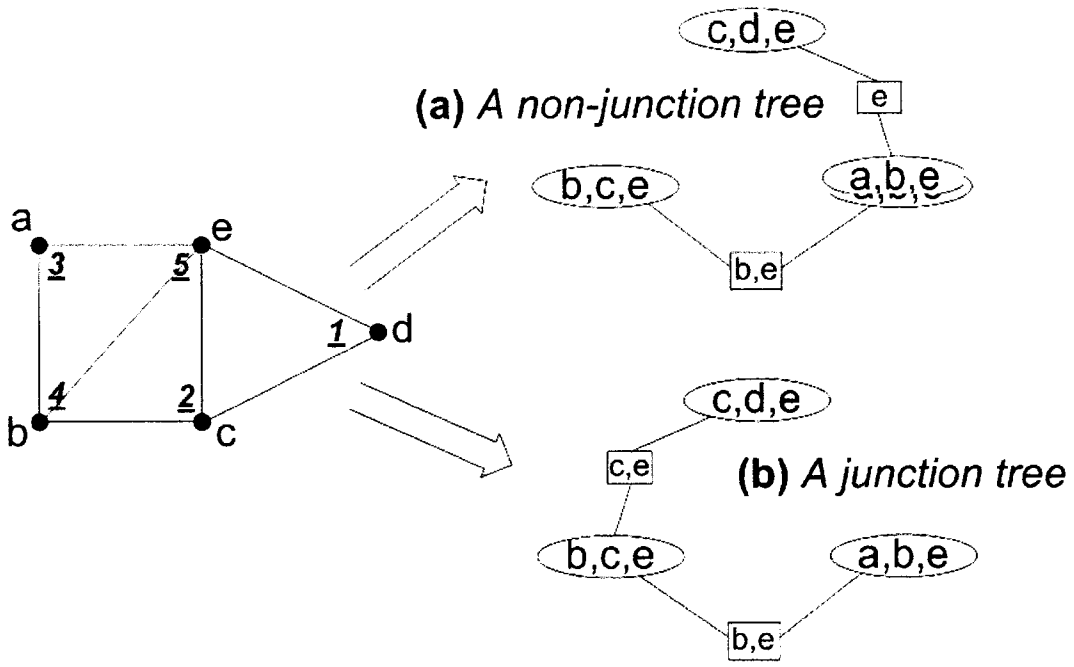


Figure 2.6: A chordal graph and two corresponding clique trees

vertices are cliques, and the links are separators. I this setting Prim's Algorithm becomes:

1. Choose a single clique arbitrarily Q^*;
2. start a junction tree T with the chosen clique Q^*;
3. add remaining cliques one at a time. At each step connect a new clique Q^* with a clique Q^T in T that has the most number of variables in common, unless this would create a cycle. If two or more separators have the same weight, add the corresponding cliques arbitrarily.

Given a set Ω of cliques $\Omega = \{\{a,b,c,d\},\ \{e,f,b,g\},\ \{e,f,h,i\},\ \{j,f,k\}\}$, Figure 2.7 illustrates applying the greedy spanning tree algorithm on Ω. The four clusters C_0 through C_3 are shown in (a). Suppose that the junction tree T starts with a single cluster C_2 arbitrarily. The intersections of C_2 with each of C_0, C_1, and C_3 are $\emptyset$, $\{e,f\}$, and f, respectively. The weights of the corresponding separators will be 0, 2, and 1, respectively. Hence, C_1 is connected to C_2 as in (b). The remaining clusters C_0 and C_3 have the intersections $\emptyset$ and $\{f\}$ with C_2, respectively, and have the intersections $\{b\}$ and $\{f\}$ with C_1, respectively. At this point, three alternatives produce the equal total weight: connecting C_3 to C_2, connecting C_0 to C_1, or connecting C_3 to C_1. Suppose that the tie is arbitrarily broken by connecting C_3 to C_2, as in (c). Finally, the remaining cluster C_0 has the intersections $\{b\}$, $\emptyset$, and $\emptyset$ with C_1, C_2, and C_3, respectively. It is connected with C_1 as in (d). Construction of the junction tree T is done.

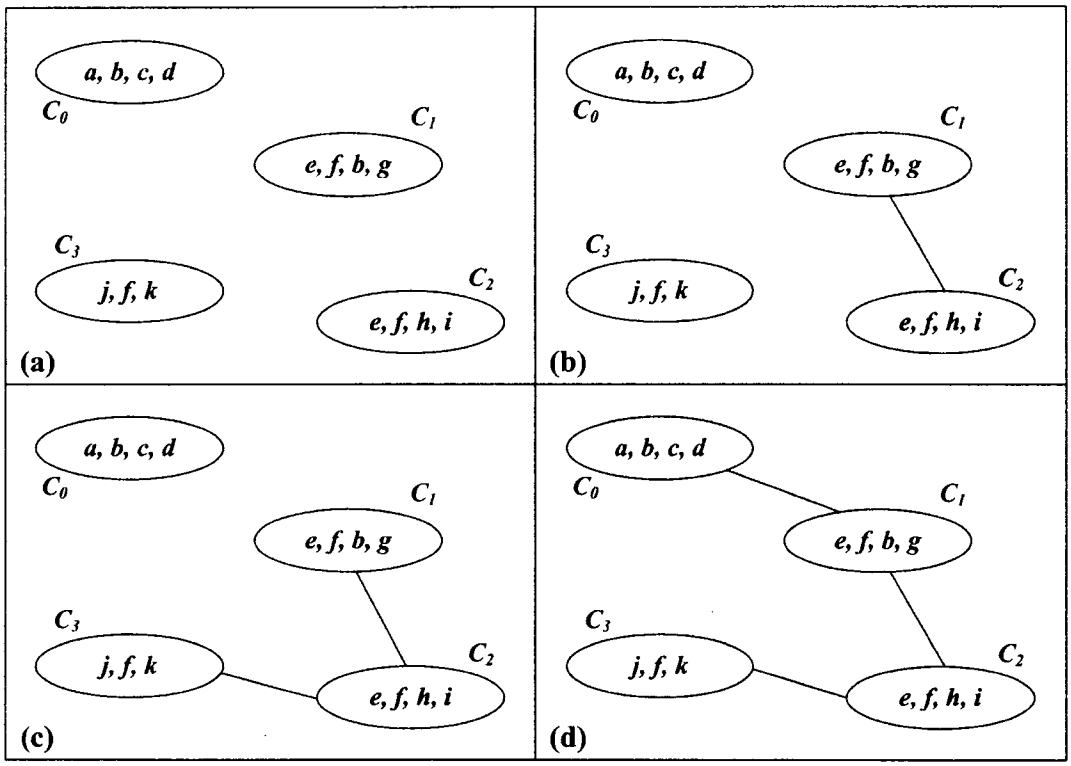


Figure 2.7: Junction tree construction

From what we just have seen in the preceding example, the construction of a junction tree is not unique. That is, given a chordal graph, multiple corresponding junction trees exist in general. However, they all satisfy the *running intersection* property.

Before we introduce the propagation part in junction trees, we need to give each separator in the JT a *label* consisting of the intersection between its terminal clusters. As we can see in Figure 2.6(b), clusters are shaped as ellipses but separators are shaped as boxes.

2.5 Belief Updating in Junction Trees

In order to use a JT representation of a BN for belief updating, the CPTs of the BN need to be converted into the *potentials* over the JT. A potential function ψ_C of a particular cluster C is defined as a function that associates to each joint realization of the random variables in C a positive real number (which is called *potential*) [3]. This potential can be considered as a non-normalized (not summing to 1) probability distribution defined over a cluster Q or its subsets. A potential is equivalent to a probability distribution because it differs only by a normalizing constant and the constant can be removed at any time [21].

2.5.1 The System Potential of a JT

The following two formulas show the relation between the JPD of a BN domain and the cluster and separator potentials of a JT representation.

Given a junction tree representation $T = (V, \Omega, \Psi)$, we define the system potential SP for T as

$$\psi_T(V) = \prod_{Q \in \Omega} \psi_Q(Q) \Big/ \prod_{S \in \Omega} \psi_S(S) \tag{2.1}$$

The system potential $\psi_T(V)$ of a JT T is equal to the JPD defined by its original BN.

$$\psi_T(V) = \prod_{v \in V} P(v|\pi(v)) = P(V) \tag{2.2}$$

2.5.2 Initialization of potentials

Once the DAG of a BN has been moralized and then triangulated, and a junction tree T has been built, The clusters in T shall have joint probability tables, known as *potentials*, attached to them. The size of each potential table is the product of the numbers of states of relevant variables. The total size increases exponentially with the sizes of the cliques.

Now, we show how to *initialize the clusters potentials* in a JT representation so that its system potential, SP, is equivalent to the JPD of the corresponding Bayesian network; see Equation 2.2.

For each probability distribution (CPT) in the DAG G of the original Bayesian network, assign this distribution into only one cluster in T that contains all the variables referenced by the CPT (The moralization step guarantees the existence of at least one such node). The following few lines have more details:
Given a junction tree $T = (V, \Omega, E)$ obtained from the DAG $G = (V, G, P)$ of a BN, for each cluster Q and each separator S, create a potential ψ_Q or ψ_S for it and initialize it to unity ($\psi_Q = 1$, $\psi_S = 1$). A potential ψ_Q is uniform if for each $q \in D_Q$, $\psi_{Q=q} = 1$. Let $fmly(v) = \{v\} \cup \pi(v)$, for each node v in G, find a cluster Q_v in T such that $fmly(v) \subseteq Q_v$ and break ties arbitrarily. Update ψ_{Q_v} to the product $\psi_{Q_v} * P(v|\pi(v))$. From the BN in Figure 1.3 on page 6 and its JT in Figure 2.1 on page 14, Figure 2.8 illustrates the potential of each cluster.

2.5.3 Consistency in a JT

Given two adjacent clusters Q and C in a junction tree T, Q and C are said to be *consistent* if their potentials, ψ_Q and ψ_C, satisfy the following formula:

$$\sum_{Q \setminus S} \psi_Q(Q) = const_1 * \psi_S(S) = const_2 * \sum_{C \setminus S} \psi_C(C) \tag{2.3}$$

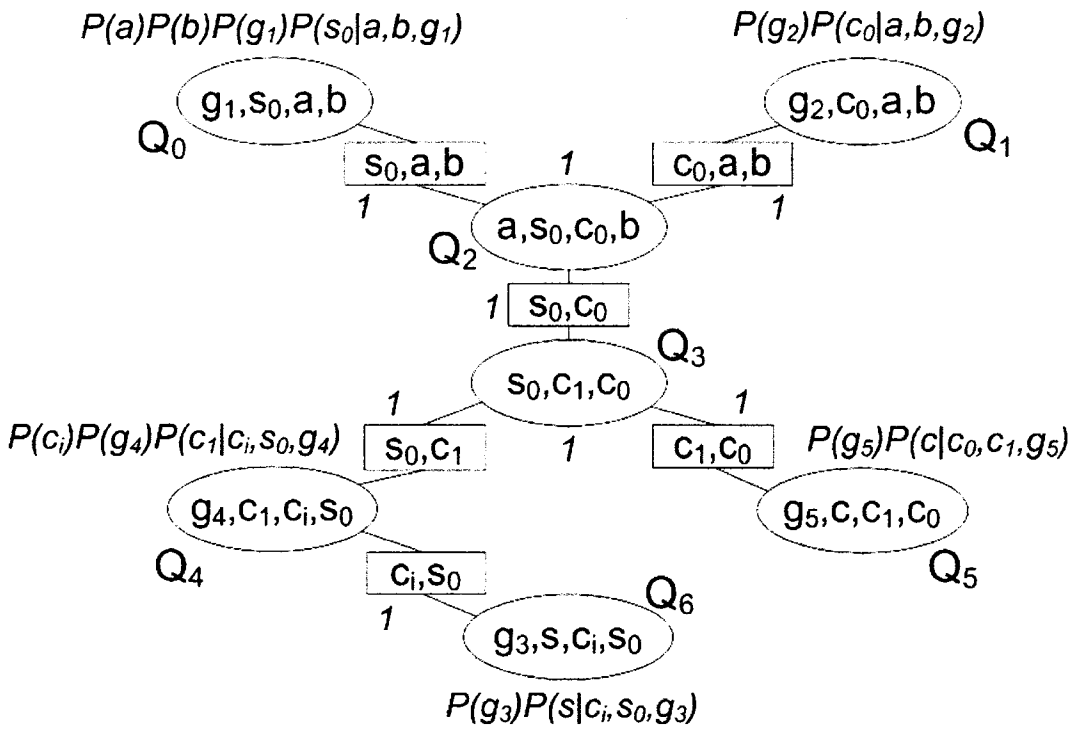


Figure 2.8: Potential assignment of each cluster in the JT of Figure 2.1.

where S is the separator between Q and C, and $const_1$ and $const_2$ stand for positive constants.

Local Consistency:

Given a junction tree $T = (V, \Omega, E, \Psi)$, if every pair of adjacent clusters $\langle Q, C \rangle$ is consistent, T is said to be *locally consistent*. Because T is locally consistent, if any cluster Q passes a potential to an adjacent cluster C over their separator $S_{\langle Q,C \rangle}$, the message cannot change $\psi_C(C)$.

Global Consistency

If every pair of clusters, (not necessarily adjacent) Q and C, is consistent, then T is said to be *globally consistent* and it holds that:

$$\sum_{Q \setminus C} \psi_Q(Q) = const * \sum_{C \setminus Q} \psi_C(C) \tag{2.4}$$

2.5.4 The Absorption Method

This is a method developed by Jensen in 1990 [11], for belief updating in a JT representation of a BN through concise message passing. The main purpose of

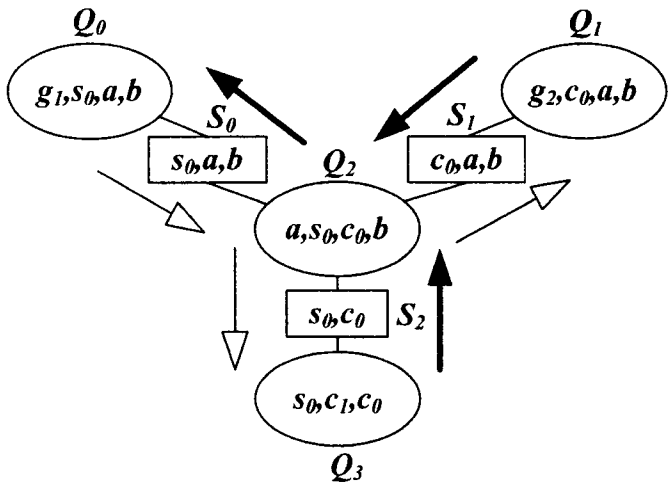


Figure 2.9: Illustration of *CollectEvidence* (the black arrows) and *DistributeEvidence* (the white arrows) activated from Q_0.

Absorption is to bring adjacent clusters in a JT representation into consistence. Absorption is an operation that passes messages from a cluster Q to an adjacent cluster C through their separator S and modifies ψ_S and ψ_C in such a way that the joint probability of the junction tree is preserved [21]. Such an operation can be done by performing the following:

1. From Q to S by updating $\psi_S(S)$ to $\psi'_S(S) = \sum_{Q \setminus S} \psi_Q(Q)$.

2. From S to C by updating $\psi_C(C)$ to $\psi'_C(C) = \psi_C(C) * \frac{\psi'_S(S)}{\psi_S(S)}$.

Given two adjacent clusters Q and C in a JT T and their separator S, Q and C are consistent after an absorption is performed over the separator in each direction (C absorbs from Q over S followed by the absorption of Q from C over S).

2.5.5 The propagation algorithm

In general, inference in JTs is not different from what we have mentioned in Section 1.5 on page 8 about inference in BNs. Again, belief updating is a process whose main goal is to compute the probability function $P(Query|Evidence)$. The evidence, in fact, is the instantiation of a variable in the probabilistic system.

Section 2.5.4 on page 23 recommends that, to make a JT $T = (V, \Omega, \Psi)$ locally consistent, absorptions need to be done along every separator in Ω in both directions. These absorptions can be organized into two cycles of message passing, *CollectEvidence* and *DistributeEvidence* [11]. Figure 2.9 illustrates these two cycles.

The following two algorithms represent the two propagation cycles *CollectEvidence* and *DistributeEvidence* respectively:

Algorithm *CollectEvidence*

Given a JT $T = (V, \Omega, \Psi)$, let Q be a cluster in T such that $Q \in \Omega$. A caller is either an adjacent cluster C or T itself. When *CollectEvidence* is called in Q, it does the following:

1. *Q calls CollectEvidence in each adjacent cluster except the caller.*
2. *After each called cluster has finished, Q absorbs from it.*

Algorithm *DistributeEvidence*

Given a JT $T = (V, \Omega, \Psi)$, let Q be a cluster in T such that $Q \in \Omega$. A caller is either an adjacent cluster C or T itself. When *DistributeEvidence* is called in Q, it does the following:

1. *If the caller is a cluster, Q absorbs from it.*
2. *Q calls DistributeEvidence in each adjacent cluster except the caller.*

From the consistency definition in Section 2.5.3 on page 22, we can guarantee that by performing *CollectEvidence* and *DistributeEvidence* on a JT T, every pair of clusters is consistent. Consequently, T is globally consistent and the potential over T is unified [21].

In Figure 2.9, T calls Q_0 to start *CollectEvidence* cycle. In response, Q_0 calls Q_2, which in turn calls Q_1 and Q_3. When Q_1 (Q_3) is called, it has no adjacent cluster except the caller and hence returns immediately, causing Q_2 to absorb from Q_1 (Q_3) through S_1 (S_2). After Q_2 finishes, Q_0 absorbs from Q_2 and terminates *CollectEvidence*. The two cycles of message passing are combined in $UnifyBelief$.

Algorithm *UnifyBelief*

Given a JT representation $T = (V, \Omega, \Psi)$, T selects a cluster $Q^*, (Q^* \in \Omega)$, arbitrarily and calls the process CollectEvidence in Q^*. After it has finished, T calls the process DistributeEvidence in Q^*.
Then after $UnifyBelief$ has finished, T is globally consistent.

2.5.6 Applying Observations

For bringing the JT T into global consistency again after an observation has been applied, we need nothing more than the three algorithms, ($UnifyBelief$, *CollectEvidence* and *DistributeEvidence*), that just have been mentioned.
All that remains is how instantiation of a variable can be applied in T and what modifications need to be done among the set Ψ of potentials to reflect that observation.

Of course some potentials of specific clusters in T have to be modified. The following algorithm, called *EnterEvidence*, shows how an observation can be applied in details.

Algorithm *EnterEvidence*

Given a JT representation $T = (V, \Omega, \Psi)$ *and an observation* $X = x$, *for each* $x \in X$, *find a cluster* Q *in* Ω *such that* $x \in Q$ *and replace* $\psi_Q(Q)$ *by* $\psi'_Q = \psi(Q) * obs(x)$. *Where* $obs(x) = \begin{cases} 1 & if X = x; \\ 0 & otherwise. \end{cases}$

After *EnterEvidence* is performed the system potential $\Psi_T(T)$ has been changed. If $UnifyBelief$ is performed next, according to Section 2.5.5, $P(Q|x)$ can be obtained from each cluster Q, *where* $X \in Q$, by normalizing its potential $\psi_Q(Q)$.

Chapter 3

Multiply Sectioned Bayesian Networks as Linked Junction Forests

3.1 Limitations of the Single-agent Paradigm

Chapter 2 studied exact probabilistic reasoning using a junction tree representation converted from a Bayesian network. It is considered that an agent makes observations on a domain, performs probabilistic inference based on its knowledge about the relations among domain events, and estimates the state of the domain. However, a single agent is limited by its knowledge, its perspective, and its computational resources. Such a locality has its limitations:

- A problem domain may be too large and complex (e.g., designing intricate machines and monitoring and troubleshooting complicated mechanisms), and thus building a single agent capable of being in charge of the reasoning task for the entire domain becomes too difficult.
- The problem domain may be open, distributed, or spread over a large geographical area, and thus transmitting observed variable values from many regions to a central location for processing is undesirable owing to communications cost, delay, and unreliability (due to communication failures).

Because of these limitations, Yang Xiang [23] has realized that a set of cooperating agents is needed to address the reasoning task effectively, that is using the locality property of large BNs to create ***multiply sectioned Bayesian networks (MSBNs)*** that are composed of one or more sections. This approach explores the natural distribution of knowledge and sensors, and to distribute modeling and inference accordingly using the *multiagent paradigm.*

Under the single-agent paradigm, a single computational entity, an agent, has access to a BN over a problem domain, converts the BN into a JT, acquires obser-

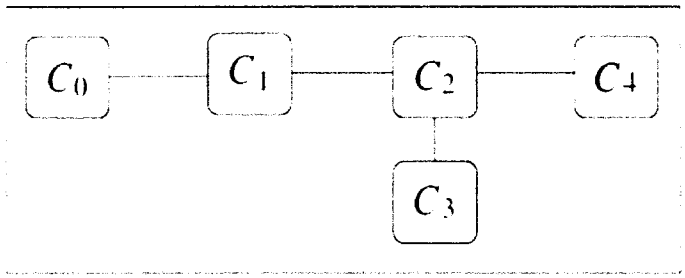


Figure 3.1: A system of five components.

vations from the domain, reasons about the state of the domain by concise message passing over the JT, and takes actions accordingly.

In this chapter we consider large and complex uncertain problem domains populated by multiple autonomous agents. Also we show that the junction tree method, presented in details in Chapter 2, can be extended to inference in multi-agent reasoning systems. Clearly, to benefit from the knowledge and observations of others, agents *must communicate* (The term *communication* refers to any exchange of messages between two or more agents).

Figure 3.1 shows a system of five components that may be remotely located. Each component is interfaced with one or more additional components.

3.2 Background Knowledge

The *MSBN technique* is an extension to the *junction tree technique* which transforms a Bayesian network into an equivalent secondary structure where inference is conducted. Because of this restructuring, belief propagation in multiply connected Bayesian networks can be performed in a manner similar to that used in singly connected networks [23].

3.2.1 Definition of hypertree structure

Let $G = (V, E)$ be a connected graph sectioned into subgraphs $\{G_i = (V_i, E_i)\}$. Let the $G_i s$ be organized as a connected tree Υ, where each node of Υ is labeled by a G_i and each link between G_m and G_n is labeled by the interface $V_m \cap V_n$ 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 Υ. Then Υ is a *hypertree* over G, each G_i is a *hypernode*, and each interface is a *hyperlink*.

Figure 3.2 shows an example hypertree.

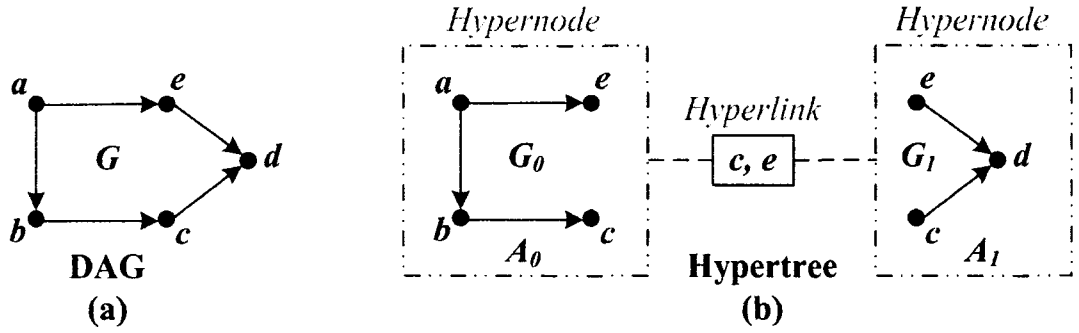


Figure 3.2: Υ in (b) is a *hypertree* over G in (a) after G has been sectioned in to two subdomains.

3.2.2 Definition of MSBN

In general, an MSBN is a knowledge representation for distributed multi-agent uncertain reasoning.

An *MSBN* M is a triplet (V, G, P): $V = \cup_i V_i$ is the total universe where each V_i is a set of variables called a *subdomain*, $G = \cup_i G_i$ is the structure, *a hypertree MSDAG*, and $P = \cup_i P_i$ 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.

3.2.3 The JPD of the MSBN

Given that for each variable $x \in V$, exactly one of its occurrences (where $\{x\} \cup \pi(x)$ is found) is assigned $P(x|\pi(x))$, and each occurrence in other subgraphs is assigned the uniform potential "1", we have

$$\begin{aligned} P(V) &= \prod_i P_i(V_i) \\ &= \prod_{x \in V} P(x|\pi(x)) \end{aligned} \tag{3.1}$$

3.2.4 D-Sepset Concept

Let G_1 and G_2 be two DAGs such that $G = G_1 \cup G_2$ is a DAG. A node $x \in I (I = G_1 \cap G_2)$ with its parents $\pi(x)$ in G is a d-sepnode between G_1 and G_2 if either $\pi(x) \in V_1$ or $\pi(x) \in V_2$. *If every $x \in I$ is a d-sepnode, then I is a* ***d-sepset.***

Figure 3.3 illustrates the d-sepset concept. The agent interface is $\{d, e\}$. For the shared variable e, we have $\pi(e) = b, d$ and $\pi(e) \in V_0$. Therefore, e is a d-sepnode.

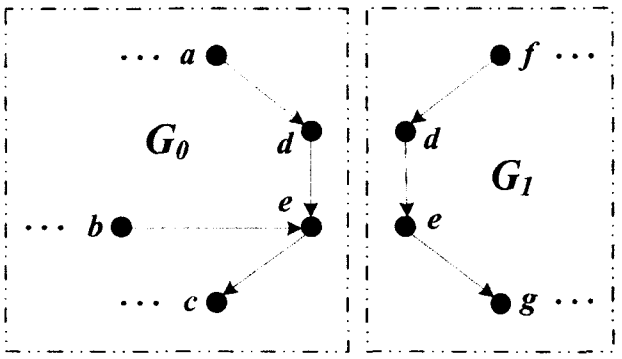


Figure 3.3: An agent interface $\{d, e\}$ that is not a d-sepset. Each box represents the DAG of one agent. Dots represent additional variables not shown explicitly.

For d, we have $\pi(d) = \{a, f\}$, $\pi(e) \not\subset V_0$ and $\pi(e) \not\subset V_1$. Therefore, d is not a d-sepnode and $\{d, e\}$ is not a d-sepset. If, however, the arc $\langle a, d\rangle$ in G_0 were reversed, then $\{d, e\}$ would be a d-sepset.

3.2.5 I-messages and E-messages

To distinguish the message sent between nodes or clusters in earlier chapters, we refer to it as an internal message or simply *i-message*, and we refer to the message sent between agents as an external message or simply *e-message*. Accordingly, we term direct communication between agents e-message passing. Note that both i-messages and e-messages are concise messages.

3.3 The Five Basic Assumptions of MSBNs

It is clear that this representation is quite restrictive, especially with respect to the allowed agent network structure. The following is a list of the major restrictions [22]: 1. Beliefs are represented using probabilities. 2. The complete domain is sectioned into subdomains, and through shared variables the agents form a connected network. 3. This agent network is structured as a tree. 4. The agent tree satisfies the running intersection property, i.e. is a hypertree. 5. Each subdomain is structured as a DAG. 6. The union of the DAGs of all subdomains is a connected DAG. 7. Each edge in the hypertree is a d-sepset. 8. The JPD over the domain is specified as in Section 3.2.3 on page 29.

In this section, it is shown that the MSBN restrictions follow directly from a set of very basic assumptions and required properties [21]. These assumptions are shown to give rise to a particular knowledge representation formalism termed *multiply sectioned Bayesian networks (MSBNs).*

1. The first basic of these stipulates that each agent's belief over its subdomain is represented by probability distribution. This assumption not only requires

each agent to represent its belief using a probability distribution but also to perform belief updating *exactly.*

Consider a total universe V of variables over which a multi-agent reasoning system made of n agents $A_0, ..., A_{n-1}$ is defined. Each agent A_i has knowledge over a subdomain V_i, where $V_i \subset V$ and $\cup_i V_i = V$. From this basic assumption, the knowledge of A_i is a probability distribution over its subdomain V_i denoted by $P_i(V_i)$.

2. The second basic assumption requires an agent A_i to communicate directly with another agent A_j only with a concise message: *its belief over the variables they share* $P_i(V_i \cap V_j)$ *where* $V_i \cap Vj \neq \emptyset$.

   The *i-message* passing in a junction tree can be used to achieve local consistence between adjacent clusters (over their separator belief). Similarly, *e-message* passing can be used to achieve consistence between adjacent agents (over their interface belief) in their tree organization.

3. The third basic assumption requires that a simpler agent organization is preferred in which agent communication by concise message passing is achievable. According to the result from Chapter 2, it follows that *a tree-organization for agent communication should be adopted.*

4. The fourth basic assumption requires each agent to represent its subdomain dependence as a DAG. As demonstrated in Chapter 1, a DAG allows the agent's belief over a subdomain to be encoded concisely (through the chain rule). Hence, this assumption is a requirement about efficiency.

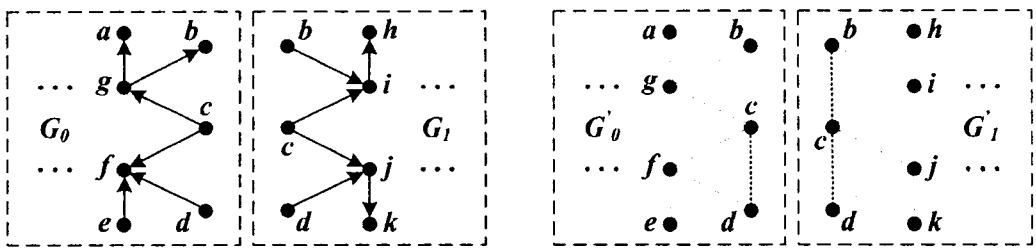
5. The last basic assumption requires the JPD to be consistent with each agent's belief over its subdomain. The assumption enforces cooperation among agents and interprets the JPD thus defined as the collective belief of all agents.

3.4 Linked Junction Forests

It is desirable that a multi-agent MSBN performs exact inference effectively by concise message passing as in the case of single-agent BNs. Chapter 2 showed how to transform or compile a multiply connected BN into a JT representation to perform belief updating by message passing. Because each subnet in an MSBN is multiply connected in general, a similar compilation is needed to perform belief updating in an MSBN by message passing.

3.4.1 The LJF Framework

In order to perform efficient inference in a distributed network, it is desirable to transform each subdomain of an MSBN into a JT representation which will stand



(a) An MSDAG consists of two sub-graphs: G_0 and G_1 *(b) The local moral graphs of G_0 and G_1*

Figure 3.4: Individual local moralization does not ensure correct moralization of a hypertree MSDAG. In (b), the link $\langle b, c\rangle$ in G'_1 is not found in G'_0 ($\because$ G'_0 and G'_1 are not *graph-consistent*).

as an inference entity. The transformation takes several steps to be done. First performin a global computation (moralization and triangulation) on the MSDAG to find a set of chordal graphs from which a set of clique hypergraphs are formed. Then the set of clique hypergraphs are organized into a set of JTs. Afterwards interfaces or d-sepsets between the JTs are created in an alternate representation called *linkage tree*. Finally, belief tables are assigned to clusters and linkages in a certain manner. The outcome of the compilation is an alternative dependence structure called a *linked junction forest*.

The message passing includes *i-message* passing within an agent for local inference as well as *e-message* passing between agents so that each can benefit from information maintained by others.

3.4.2 Cooperative Distributive Moralization of MSDAG

The first step in compilation of an MSBN is to transform the structure of its hypertree MSDAG into its moral graph. Recall from Section 2.2 on page 15 that the moral graph of a DAG is obtained by connecting parents pairwise for each child node and dropping the direction of each arc.
Because local computation by individual agents without interaction does not ensure correct moralization of a hypertree MSDAG (See Figure 3.4), a cooperative distributive moralization is needed.

Recursive algorithms for each agent are presented [21]. The execution of each algorithm by an agent is activated by a call from an entity known as the *caller*. We denote the agent called to execute the algorithm by A_*. The caller is either an adjacent agent of A_* in the hypertree denoted by A_c or the system coordinator.

Algorithm (*CoMoralize*)

Algorithm *CoMoralize* is executed by the system coordinator to activate cooperative moralization.

start < *CoMoralize* >
choose an agent A_* *arbitrarily;*
call A_* *to run* ***CollectMLinks****;*
call A_* *to run* ***DistributeMLinks****;*
end < *CoMoralize* >

Algorithm (*CollectMLinks*)

An Agent A_* performs local moralization (as it is described in details in Section 2.2 on page 15), updates its moral graph with links collected from adjacent agents, and sends all relevant added moral links to the caller.

start < *CollectMLinks* >
set $MLinks = \emptyset$;
moralize G_* *and denote added moral links by* L_*;
add L_* *to* $MLinks$;
for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
call A_i *to run* ***CollectMLinks*** *and receive links* L_i *over* I_i *from* A_i *when done;*
add L_i *to* G_* *and to* $MLinks$
if caller is an agent A_c*, send* A_c *the restriction of* $MLinks$ *to* I_c
end < *CollectMLinks* >

Algorithm (*DistributeMLinks*)

An Agent A_* receives moral links from the caller A_c, updates its moral graph accordingly, and sends all relevant added moral links to each adjacent agent.

start < *DistributeMLinks* >
if caller is an agent A_c *do*
receive a set L_c *of links over* I_c *from* A_c;
add L_c *to* G_* *and to* $MLinks$;
for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
send A_i *the restriction of* $MLinks$ *to* I_i *with links in* L_i *removed;*
call A_i *to run* ***DistributeMLinks****;*
end < *DistributeMLinks* >

3.4.3 Linkage Trees as Communication Channels

A JT should not just support effective local inference, but it should also support communication among the hypertree.

An alternate representation of the agent interface, called a *linkage tree (LT)*, is used to support concise interagent message passing (e-messages). The need to construct linkage trees imposes additional constraints when the moral graph structure is triangulated into the chordal graph structure. An obvious method is to construct

the JT so that each d-sepset I with an adjacent agent is contained in a single cluster Q. The e-message will then be the potential $\psi(I)$ computed by marginalization of the cluster potential $\psi_Q(Q)$.

In large problem domains, the d-sepsets are much larger. Applying the preceding method will increase belief complexity $|\psi_Q(Q)|$ exponentially on the cardinality of d-sepsets. It is possible to generate the e-message from a less complex d-sepset by using a *cluster tree* as a graphical structure for computing the e-message.

Given the MSDAG in Figure 3.5 (a), the two corresponding JTs in (c) are equivalent to those in (d). From (d), potentials, or e-message, over d-sepset (b, c, d) can be computed in a direct way, such that

$$\begin{aligned} P(b,c,d) &= \psi_{Q_1}(Q_1) \\ &\quad or \\ &= \psi_{C_1}(C_1) \end{aligned}$$

with belief complexity

$$|P(b,c,d)| = 8$$

On the other hand, potentials over the same d-sepset can also be computed from (c) such that

$$\begin{aligned} P(b,c,d) &= P(b,c)P(c,d)/P(c) \\ &= \psi_{Q_0}(b,c) * \psi_{Q_1}(c,d)/P(c) \\ &\quad or \\ \psi_{C_0}(b,c) * \psi_{C_1}(c,d)/P(c) & \end{aligned}$$

where each of $P(b, c)$ and $P(c, d)$ is a potential over a cluster, and $P(c)$ is the potential over the corresponding separator.

The next algorithm shows how a linkage tree L can be constructed from a JT T by using a graphical operation called *merge*. Let us first introduce the *merge* operation. A cluster C in a JT is merged into another cluster Q if Q is replaced by $Q' = Q \cup C$, C is removed from the JT (together with its separator with Q), and the other clusters originally adjacent to C are made adjacent to Q'.

Algorithm (*ConstructLinkageTree*)

Given a subgraph G in a hypertree MSDAG, the d-sepset I between G and an adjacent subgraph G', and a JT T derived from G, perform the following procedure in T. The resultant cluster tree will be the linkage tree L of G with respect to G'. Figure 3.6 illustrates how the *merge* operation leads to linkage tree construction.

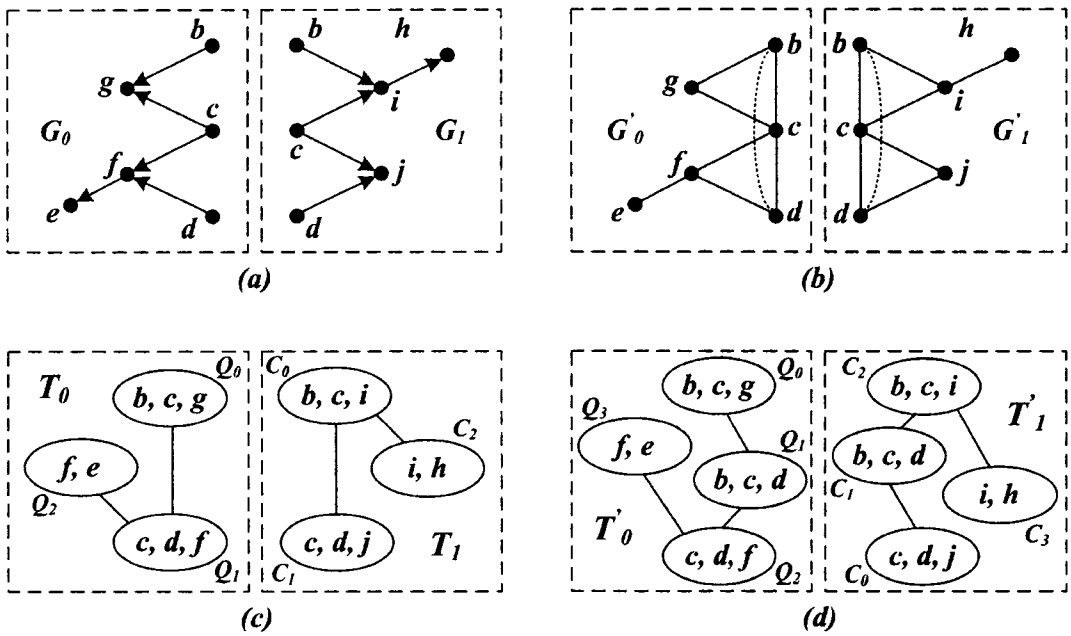


Figure 3.5: (a) An MSDAG consists of two subnets. (b) Their local moral graphs. (c) JTs constructed from local moral graphs. (d) JTs constructed after adding link $\langle b, d \rangle$ (the dotted link) to the local moral graphs.

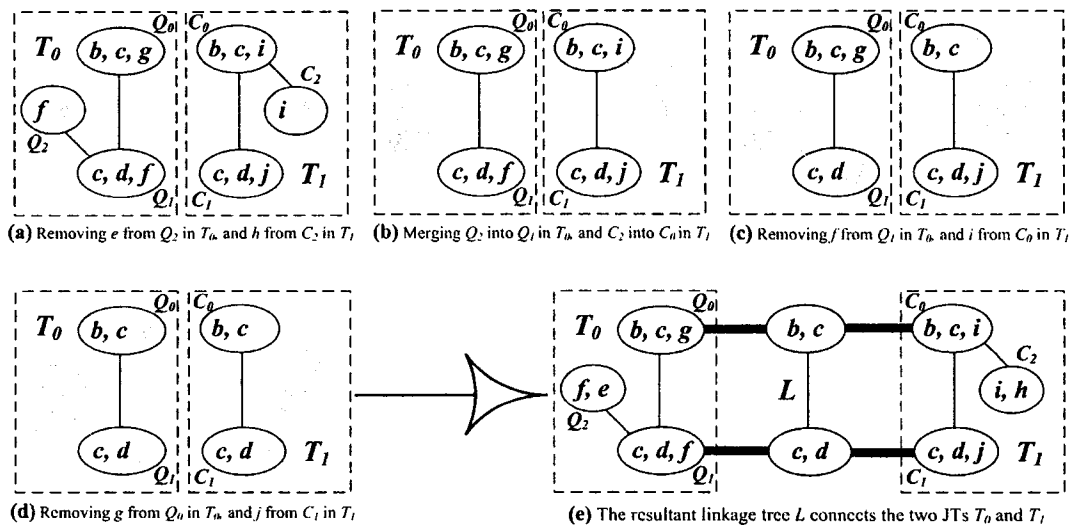


Figure 3.6: Illustration of constructing LTs using *merge* operation.

<start>
do
search for a variable x in T such that $x \notin I$;
if x is found and contained in a unique cluster C
remove x from C;
if C becomes a subset of an adjacent cluster Q ($C \subseteq Q$)
merge C into Q:
make the other clusters originally adjacent to C adjacent to Q;
remove C and its separator with Q;
until no such x found;
<end>

Let L be the resultant cluster graph. Then L is a **linkage tree** of T with respect to I if

$$\bigcup_{Q \in L} Q = I$$

where each cluster Q in L is called a ***linkage***. A cluster in T that contains Q (breaking ties arbitrarily) is called the ***linkage host*** of Q.

3.4.4 Cooperative Triangulation in a Hypertree

This step is to triangulate the moral graph of the MSDAG into a *chordal supergraph*. The chordal supergraph is the union of a set of local chordal graphs, each of which is a supergraph of a local moral graph. It is necessary for the same reason that triangulation is used when compiling a BN: *a junction tree of a graph exists if and only if the graph is chordal* [21].

Given the moral graph $G = \cup_i G_i$ of a hypertree MSDAG, each subgraph G_i over V_i needs to be converted into a chordal supergraph G_i' over V_i such that $\cup_i G_i'$ is a chordal supergraph G' of G and each pair of G_i' and G_j' is graph-consistent.

In an LJF, d-sepsets need to be formed as LTs. In order to form each d-sepset I_i of an agent A_* in a linkage tree representation, the local graph G_* of A_* needs to be chordal supergraph G_*'. In other words, G_*' must be eliminatable with respect to each adjacent agent A_i in the hypertree Υ in the order $(V_* \backslash I_i, I_i)$, *where* $(i = 1...adj(A_*))$.

Next we present recursive algorithms for each agent for cooperative multi-agent triangulation considering the most general case: to triangulate the moral graph of a general hypertree MSDAG when the hypertree is populated by $n > 3$ agents. The execution of each algorithm by an agent (denoted by A_*) is activated by a caller, which is either an adjacent agent (denoted by A_c) of A_* or the system coordinator.

Algorithm (*CoTriangulate*)

CoTriangulate is executed by the system coordinator to activate the cooperative triangulation by multiple agents.

start < *CoTriangulate* >
  choose an agent A_* *arbitrarily;*
  call A_* *to run* ***DepthFirstEliminate****;*
  after A_* *has finished, call* A_* *to run* ***DistributeCLinks****;*
end < *CoTriangulate* >

Algorithm (*DepthFirstEliminate*)

An Agent A_* performs *triangulation by elimination* and updating with respect to all its adjacent agents.

start < *DepthFirstEliminate* >
  if caller is an agent A_c*, do*
    receive a set L_c *of* $fill\_ins$ *over* I_c *from* A_c*;*
    add L_c *to* G_**;*
  set $CLinks = \emptyset$*;*
  for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
    eliminate V_* *in the order* $(V_* \backslash I_i, I_i)$ *and denote the resultant* $fill\_ins$ *by* L*;*
    add L *to* G_* *and to* $CLinks$*;*
    send A_i *the restriction of* $CLinks$ *to* I_i*;*
    call A_i *to run* ***DepthFirstEliminate*** *and receive* $fill\_ins$ L_i *over* I_i *from* A_i*;*
    add L_i *to* G_* *and to* $CLinks$*;*
  if caller is an agent A_c*, do*
    eliminate V_* *in the order* $(V_* \backslash I_c, I_c)$ *and denote the resultant* $fill\_ins$ *by* L*;*
    add L *to* G_* *and to* $CLinks$*;*
    send A_c *the restriction of* $CLinks$ *to* I_c*;*
end < *DepthFirstEliminate* >

Algorithm (*DistributeCLinks*)

An Agent A_* receives moral links from the caller A_c, updates its moral graph accordingly, and sends all relevant added moral links to each adjacent agent.

start < *DistributeCLinks* >
  if caller is an agent A_c *do*
    receive a set L_c *of* $fill\_ins$ *over* I_c *from* A_c*;*
    add L_c *to* G_* *and to* $CLinks$*;*
  for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
    send A_i *the restriction of* $CLinks$ *to* I_i*;*
    call A_i *to run* ***DistributeCLinks****;*
end < *DistributeCLinks* >

Because only one execution of *CoTriangulate* does not guarantee that each G_i ($G = \cup_i G_i$) is eliminatable with respect to all of its adjacent agents in the order $(V_i \backslash I_j, I_j)$, it is required to rerun CoTriangulate until no new *fill_ins* are produced. The following algorithm *SafeCoTriangulate* extends *CoTriangulate* to ensure the requirement on eliminatable order is fully satisfied.

Algorithm (*SafeCoTriangulate*)

CoTriangulate is executed by the system coordinator to activate the safe cooperative triangulation by multiple agents.

start < *SafeCoTriangulate* >
do
perform ***CoTriangulate;***
each agent performs an elimination relative to the d-sepset with each adjacent agent;
until no agent added any fill_in;
end < *SafeCoTriangulate* >

3.4.5 Constructing Local JTs and Linkage Trees: (LJF)

Before inference in a hypertree MSDAG is possible the structure has to be transformed into an LJF with LTs as communication channels before belief updating through concise message passing is possible [23] [21].

The chordal supergraph needs to be organized into an LJF representation to do effective inference with concise message passing. This can be performed by:

- first, organizing all the local chordal supergraphs into their JT representation. This transformation can be done locally in each agent without any cooperation from any adjacent agent.
- second, because each JT (as an entity) in an LJF is connected with its adjacent JTs through LTs, those linkage trees must be constructed locally from the agent's JT using the method described in Section 3.4.3 on page 33.

This task needs to be performed by each individual agent. The final structure will be an LJF.

Figure 3.7 illustrates how a MSDAG of a full-adder digital circuit can be compiled in to a multi-agent LJF managed by two agents.

After the local JT and LTs are constructed by each agent, the hypertree MSDAG in the original MSBN has been converted into a different dependence structure, which is an LJF. Thus, an LJF structure is formally defined as follows:

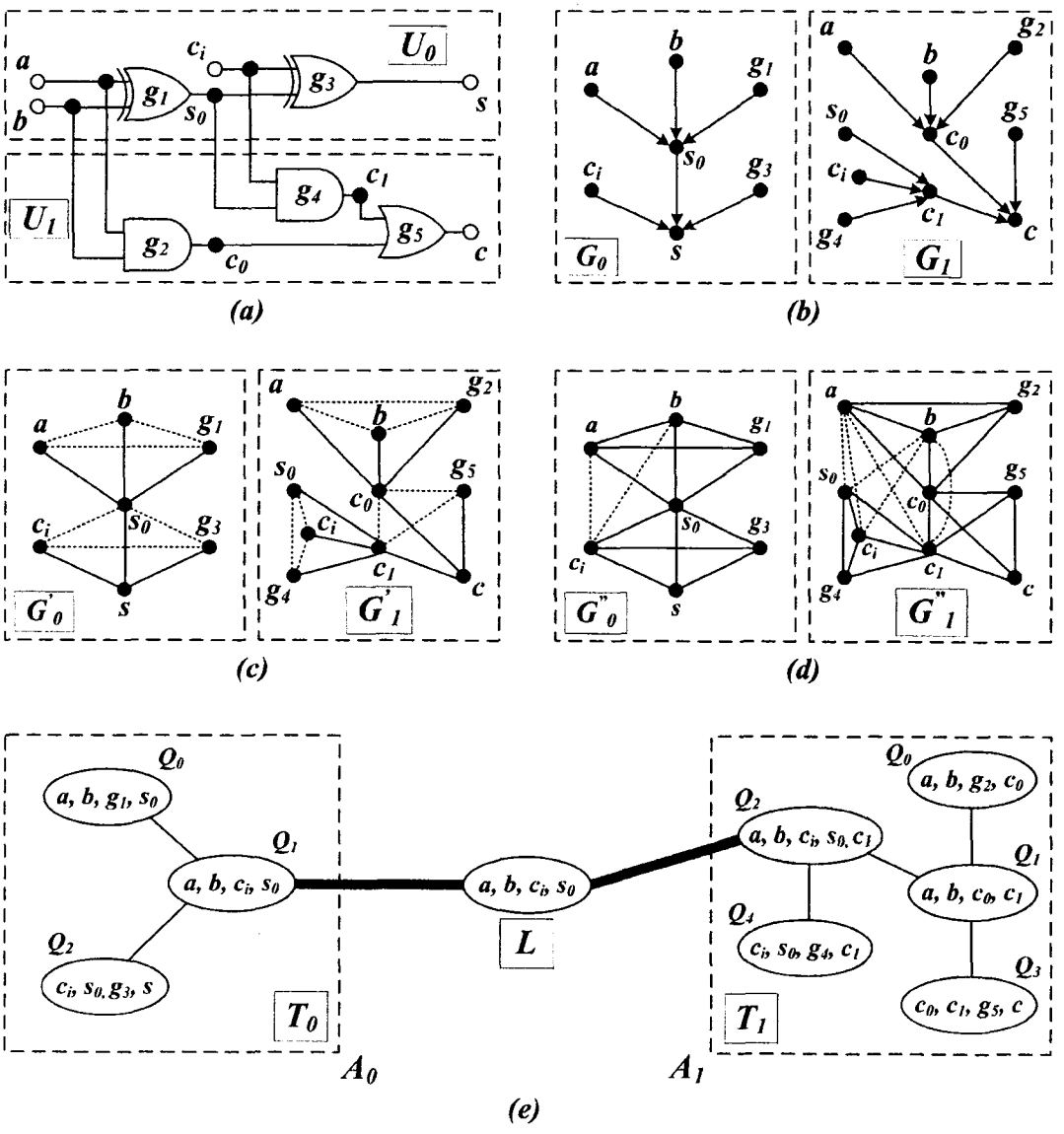


Figure 3.7: (a): A full-adder digital circuit consists of two units. (b): The MSDAG of (a). (c) and (d): The resultant moral and chordal supergraphs respectively. (e): The LJF representation consists of two agents each with a copy of L.

An LJF F is a tuple (V, G, T, L). $V = \cup_i V_i$ is the total universe, where each V_i is a set of variables, called a subdomain.

$G = \cup_i G_i$, where each $G_i = (V_i, E_i)$ is a chordal graph such that there exists a hypertree Υ over G.

$T = \{T_i\}$ is a set of JTs, each of which is a JT of the corresponding G_i.

$L = \{L_i\}$ is a collection of LT sets. Each $L_i = \{L_{i,j}\}$ is a set of linkage trees, one for each hyperlink incident to G_i in Υ. Each $L_{i,j}$ is a LT of T_i with respect to a hyperlink $V_i \cap V_j$.

Chapter 4

Inference in Distributed Multi-agent Reasoning Systems

In order to start performing belief updating in an LJF representation, the quantitative knowledge in the original MSBN needs to be transformed into knowledge in the LJF as well. As in the single-agent paradigm, the knowledge needs to be converted from conditional probability distributions over the variables in the subnet into potentials over the cliques and separators in the corresponding JT. Because in an LJF we have additional entities, linkage trees, these too have to be assigned potentials.

4.1 Initial Potential Assignment

In each JT T_i in the LJF, in the same way as in the single-agent paradigm, first, each cluster and each separator in T_i is given a uniform potential "1". Second, for each variable x find a cluster Q that contains $fmly(x)$ ($fmly(x) \subseteq Q$), and break ties arbitrarily. Update $\psi_Q(Q)$ to $\psi'_Q(Q) = \psi_Q(Q) * P(x|\pi(x))$. In each LT L_{ij} in T_i, each linkage in L_{ij} is given a uniform potential.

Once potentials for clusters $\psi_{Q_j}(Q_j)$ and separators $\psi_{S_k}(S_k)$ of local JTs T_i and linkages $\psi_{Q_l}(Q_l)$ of LTs are assigned, all other potentials can be defined accordingly. These include the following potentials:

- a potential for each local JT:

$$\psi_{T_i}(V_i) = \Big[\prod_j \psi_{Q_j}(Q_j)\Big] \Big/ \Big[\prod_k \psi_{S_k}(S_k)\Big] \tag{4.1}$$

  where j is over the indices of all clusters in T_i and k is over the indices of all separators.

- a potential for each separator in each LT. A potential of a separator S_k can be computed from either Q_k of the two linkages with it:

$$\psi_{S_k}(S_k) = \sum_{Q_k \setminus S_k} \psi_{Q_k}(Q_k) \tag{4.2}$$

- a potential for each LT. Because a LT is a tree, its potential can be calculated just as in a JT and is defined as:

$$\psi_{L_{i,j}}(I_{i,j}) = \Big[\prod_k \psi_{Q_k}(Q_k)\Big] \Big/ \Big[\prod_l \psi_{S_l}(S_l)\Big] \tag{4.3}$$

  where k is over the indices of all clusters in $L_{i,j}$ and l is over the indices of all separators.

- a joint system potential (JSP) for the entire LJF representation. The JSP over the universe V is associated with the LJF F and is defined as

$$\Psi_F(V) = \Big[\prod_i \psi_{T_i}(V_i)\Big] \Big/ \Big[\prod_j \psi_{L_j}(I_j)\Big] \tag{4.4}$$

  where i is over the indices of all JTs in F and j is over the indices of linkage tree (one for each d-sepset).

4.2 E-message Passing among Agents

4.2.1 Why Extended Linkage Potential?

Recall from Equation 4.3 that the potential of a linkage tree $L_{i,j}$ over a d-sepset $I_{i,j}$ is defined as

$$\psi_{L_{i,j}}(I_{i,j}) = \Big[\prod_k \psi_{Q_k}(Q_k)\Big] \Big/ \Big[\prod_l \psi_{S_l}(S_l)\Big]$$

where k is over the indices of all linkages in $L_{i,j}$ and l is over the indices of all separators.

When the LT is locally consistent, each separator potential carries redundant information because it is simply a marginal of some cluster potential. The appearance of the product of separator potentials as a denominator in the preceding equation is just to *remove* this redundance from the numerator (the product of cluster potentials).

Instead of removing redundance after the product, an *extended potential* associated with each linkage in an LT leads to a computation shortcut.

Let L be a linkage tree of a local JT. Convert L into a rooted tree by selecting a linkage Q arbitrarily as the root and direct links away from it. For each linkage

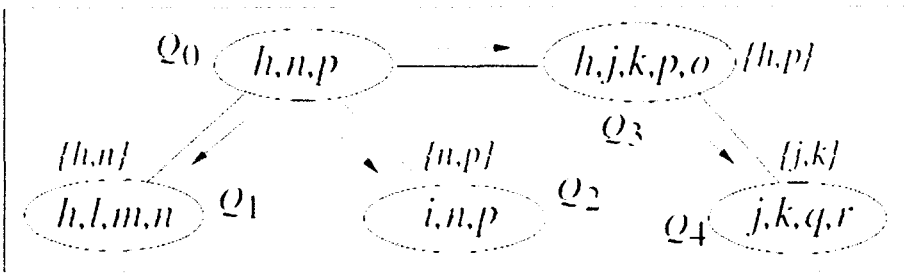


Figure 4.1: Defining linkage peers for each linkage.

$Q' \neq Q$, *assign the separator with its parent linkage as the* ***peer separator*** R *of* Q'.

In Figure 4.1, if the linkage Q_0 is chosen as the root, then no peer will be assigned to it. The peer separator of each other linkage is marked beside the linkage.
After linkage potentials, separator potentials, and linkage peers have been defined, we can derive an extended linkage potential:

for each linkage Q *with peer* R *in* L, *do*
  define its extended linkage potential $\psi^*_Q(Q) = \psi_Q(Q)/\psi_R(R)$;
for the linkage Q *labeled as a root (without peer),*
  define its extended linkage potential $\psi^*_Q(Q) = \psi_Q(Q)$;

Now the linkage tree potential $\psi_{L_{i,j}}(I_{i,j})$ can be expressed in terms of extended linkage potentials as

$$\psi_{L_{i,j}}(I_{i,j}) = \prod_Q \psi^*_Q(Q) \tag{4.5}$$

4.2.2 Passing Beliefs through Linkages

Because agents in a system organized as an LJF representation have to be consistent, we introduce two algorithms *UpdateBelief* and *AbsorbThroughLinkage* to achieve consistence by message passing [21, 23]. *UpdateBelief* algorithm is analogous to *Absorption* in Section 2.5.4 on page 23 for message passing in a single-agent JT. It propagates belief from an agent to an adjacent agent through linkages in the linkage tree between them. *AbsorbThroughLinkage* algorithm is to propagate belief from one agent to an adjacent agent through a single linkage.

Algorithm *UpdateBelief*

Here, the message originates from a local JT and is targeted at another local JT. The channel is a d-sepset and (a hyperlink) in the form of LT.

Let A_i and A_j be adjacent agents. A_i is associated with local JT T_i and LT L_i, and A_j with T_j and L_j. When UpdateBelief is called on A_i with respect to A_j, it does the following:

start < UpdateBelief >
call A_j to assign a new potential $\psi_{Q_j}(Q_j)$ for each linkage Q_j in L_j;
for each linkage Q_i with host C_i in L_i, do
call AbsorbThroughLinkage for C_i to absorb through Q_i;
call UnifyBelief (Section 2.5.5 on page 25) at T_i;
end < UpdateBelief >

Upon request from A_i to assign a new potential for each linkage in L_j, A_j does the following:

start < subUpdateBelief >
for each linkage Q_j with host C_j in L_j, do
assign $\psi_{Q_j}(Q_j) = \sum_{C_j \setminus Q_j} \psi_{C_j}(C_j)$;
end < subUpdateBelief >

Algorithm *AbsorbThroughLinkage*

Let A_i and A_j be adjacent agents. A_i is associated with the local JT T_i and linkage tree L_i, and A_j with T_j and L_j. Let Q_i be a linkage in L_i, C_i be the linkage host of Q_i in T_i, and Q_j be the corresponding linkage in L_j. When AbsorbThroughLinkage is called on A_i for C_i to absorb through Q_i, the following occurs:

start < AbsorbThroughLinkage >
send a request to A_j to transmit $\psi_{Q_j}(Q_j)$;
upon receipt, A_i updates its host potential $\psi'_{C_i}(C_i) = \psi_{C_i}(C_i) * \psi^*_{Q_j}(Q_j)/\psi^*_{Q_i}(Q_i)$;
update linkage potential $\psi^{*'}_{Q_i}(Q_i) = \psi^*_{Q_j}(Q_j)$;
end < AbsorbThroughLinkage >

4.3 The Communication Protocol in an LJF

An LJF representation is operated by multiple agents with one at each hypernode. The multi-agent belief communication in an LJF is organized as follows: The main algorithm *CommunicateBelief* is for belief propagation and calls first the algorithm *CollectBelief* and after that the algorithm *DistributeBelief* [23] [21]. *CollectBelief* recursively propagates belief inwards from terminal agents towards an initiating agent, where *CollectBelief* is analogous to *CollectEvidence* for the single-agent paradigm proposed by [11] (See Section 2.5.5 on page 24). *DistributeBelief* recursively propagates belief outwards from the initiating agent to the terminal agents and is analogous to *DistributeEvidence* also proposed

by [11]. Running *CommunicateBelief* will bring the LJF into global consistence. When observations made by the agents, the evidence is injected into the multi-agent reasoning system by the observing agent with *EnterEvidence*. After *CommunicateBelief* is called, the LJF will be returned to a globally consistent state.

The execution of each algorithm by an agent is activated by a call from an entity known as the *caller*. We denote the agent called to execute the algorithm by A_*. The caller is either an adjacent agent of A_* in the hypertree denoted by A_c or the system coordinator.

Algorithm *CommunicateBelief*

Algorithm *CommunicateBelief* is executed by the system coordinator to activate global communication.

start < *CommunicateBelief* >
  choose an agent A_* *arbitrarily;*
  call A_* *to run* ***CollectBelief****;*
  call A_* *to run* ***DistributeBelief****;*
end < *CommunicateBelief* >

Algorithm *CollectBelief*

As it is mentioned at the beginning of this section, *CollectBelief* algorithm is responsible for performing the first belief propagation cycle inwards on the hypertree of an LJF. An agent A_* updates its belief over the shared variables with all of its adjacent agents (except A_c if applicable) by absorbing through its linkage trees with them and unifies its local belief (just as an independent JT) after each absorbtion. If no adjacent agent exists except the caller, it just unifies its belief over T_*. A caller is either an adjacent agent A_c or the system coordinator.

start < *CollectBelief* >
  if there is no adjacent agent except caller
    execute ***UnifyBelief*** *locally at* T_**;*
  else
    for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
      call A_i *to run* ***CollectBelief****;*
      execute ***UpdateBelief*** *relative to* A_i*;*
end < *CollectBelief* >

Algorithm *DistributeBelief*

DistributeBelief algorithm is responsible for performing the other belief propagation cycle outwards on the hypertree of an LJF. An agent A_* updates its belief over the shared variables with the caller (if applicable) by absorbing through its

linkage tree with A_c and unifies its local belief (just as an independent JT). A_* then motivates the remaining adjacent agents to recursively distribute their beliefs. A caller is either an adjacent agent A_c or the system coordinator.

start < *DistributeBelief* >
  if caller is an agent A_c
    execute ***UpdateBelief*** *locally at* T_* *relative to* A_c;
  for each adjacent agent $A_i (i = 1, ..., n)$ *except caller, do*
    call A_i *to run* ***DistributeBelief***;
end < *DistributeBelief* >

4.3.1 Complexity of Multi-agent Communication

We consider the time complexity of ***CommunicateBelief*** in a multi-agent reasoning system organized into an LJF. We used the following parameters to characterize the LJF.

- a: the total number of agents in the system.
- b: the maximum number of clusters in a local JT.
- c: the cardinality of the largest cluster in local JTs.
- d: the cardinality of the largest junction in local JTs.
- e: the maximum number of linkages in a linkage tree.

During **CommunicateBelief** algorithm, **UpdateBelief** is performed twice for each hyperlink in the hypertree, once during **CollectBelief**, and once during **DistributeBelief**. Hence, **UpdateBelief** is performed $2(a-1)$ times because a tree of a nodes has $a-1$ links.

UpdateBelief has three steps. The first step updates up to e local linkage potentials (one for each linkage) and has a complexity of $O(e2^c)$. In the second step, **AbsorbThroughLinkage** is performed up to e times (one for each linkage). For each **AbsorbThroughLinkage**, an extended linkage potential is transmitted, and a linkage host potential is updated. Hence, this step has the complexity $O(e2^c)$. In the last step of **UpdateBelief**, **UnifyBelief** is performed. **UnifyBelief** calls **Absorptions** twice over each separator. The complexity of **Absorptions** is $O(2^c)$. Hence, complexity of **UnifyBelief** is linear on b (there are $n-1$ separators) and exponential on c, $O(b2^c)$. The overall complexity of **UpdateBelief** is then

$$O\big((b+2e)2^c\big).$$

Combining the preceding analysis, we arrive at the conclusion that the complexity of **CommunicateBelief** is

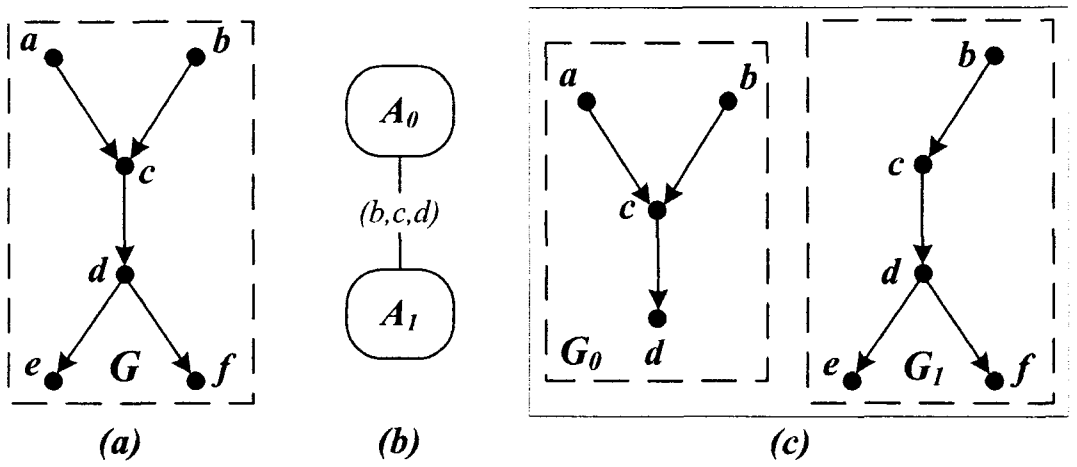


Figure 4.2: (b) A hypertree over G in (a), which is sectioned into subgraphs in (c).

$$O\big(a(b+2e)2^c\big). \tag{4.6}$$

It is assumed that the shared variables between a pair of agents are a small subset of either subdomain involved. This implies $b \gg e$. Therefore, the complexity of **CommunicateBelief** can be simplified to

$$O(ab2^c). \tag{4.7}$$

4.4 A Practical Issue on Implementing Distributed Multi-agent Reasoning Systems

The dependence structure of a multi-agent reasoning system is a hypertree multiply sectioned DAG G. Hence, a JPD over a set of variables V can be defined by specifying a local distribution for each node (or variable) and applying the chain rule. Each node in G is either internal to an agent (a non-d-sepnode), or it is shared between some agents, which in the latter case is referred to as a *d-sepnode.*

The distribution for a non-d-sepnode can be specified by the corresponding agent vendor or owner. On the other hand, the parent set of a d-sepnode within each agent may differ. In Figure 4.2(c), c has two parents in G_0 and one parent in G_1. From Section 3.2.4 on page 29, all parents $\pi(x)$ of a d-sepnode x must exist in at least one agent in G.

When agents that contain x are developed by the same vendor and owned by the same owner, only $P(x|\pi(x))$ needs to be specified. For each agent A_i that contains $\pi_i(x) \subset \pi(x)$, $P_i(x|\pi_i(x))$ is implied [21].

In a distributed multi-agent reasoning system with agents that are built by different vendors or owned by different owners, it is possible that $P_i(x|\pi_i(x))$ and

$P_j(x|\pi_j(x))$ are inconsistent for a pair of agents A_i and A_j. This may occur even when $\pi_i(x) = \pi_j(x)$. For instance, in Figure 4.2(c), $P_0(d|c)$ by A_0 may be inconsistent with $P_1(d|c)$ by A_1. Inconsistence may also occur when $\pi_i(x) \neq \pi_j(x)$. For instance, in Figure 4.2(c), A_0 may have $P_0(c = c_0|a = a_1, b = b_0) = 0.8$, whereas A_1 may specify $P_1(c = c_0|b = b_0) = 0$.

Xiang, in [23], made a basic assumption to integrate independently built agents into a coherent system. In a distributed multi-agent reasoning system, each agent has no knowledge beyond its shared variables. For example, no agent can be absolutely certain of exactly how many parents $|\pi(x)|$ a d-sepnode x has. We must also assume that each agent protects the knowledge about its non–d-sepnodes as much as possible, revealing as little as possible.

Here, we propose an efficient method that can guarantee performing belief assignment over all d-sepnodes in a hypertree. The method aims to enforce the constraint of Xiang's fifth assumption in a practical manner. The method is built mainly on the following five agreements:

1. Each node in each agent has an attribute of its nature (*true*: x is a physical local variable, *false*: otherwise).

2. Each agent has its own *rank* given by the system coordinator.

3. An agent having the maximum number of parents $|\pi(x)|$ of a d-sepnode x will be permitted to assign its conditional probability over x.

4. In case of more than one agent having the same value of $|\pi(x)|$, the agent with positive physical attribute of x is more entitled to assign its conditional probability over x.

5. When more than one agent has the same $|\pi(x)|$ and x is virtual to all of them, the agent with better rank (a lower rank) will be permitted to assign x's conditional probability.

This method uses a very simple logic. That is, an agent A_i proposes to assign its conditional probability $P_i(x|\pi_i(x))$ over a d-sepnode x (even if $|\pi_i(x)| = 0$), so it sends a request to each adjacent agent A_j asking it to assign a uniform "1" instead of its conditional probability $P_j(x|\pi_j(x))$ over x. A_j may agree or disagree. It agrees by sending back an *acceptance* and disagrees by sending back an *objection* explaining why. Through passing a sequence of such e-messages among agents organized in an LJF, a single assignment of d-sepnodes' conditional probabilities in the distributed multi-agent reasoning system can be achieved. This relaxes the condition that the JPD of each d-sepnode will be consistent with each agent's belief over its subdomain

The method is represented by three algorithms, **BeReadyToCommunicate**, **UniformV**, and **WithdrawV**. **BeReadyToCommunicate** is the main algorithm

and it has been given this identification name because it needs to be performed just before the first call of **CommunicateBelief**.

Before we present the algorithms, for each hypernode in the hypertree, assume that we have:
$I = \cup_i I_i$; *where* $i \in |adjacent\ agents|$
Rank: is given by the system coordinator;
$Assigned = \{V\} \backslash \{I\}$;
$Suspended = \emptyset$.

Algorithm *BeReadyToCommunicate*

BeReadyToCommunicate algorithm is responsible for performing the potential assignment globally for the d-sepsets' variables in the hypertree of an LJF. Before an agent A_* assigns its belief over a shared variable v, it calls its adjacent agents to assign a uniform belief over v. If any of them feels more entitled than A_*, it sends back an objection. As a response, A_* will be content with assigning a uniform belief.

The caller is either an adjacent agent A_c or the system coordinator. First, the system coordinator chooses an agent $A*$ arbitrarily and calls it to run *BeReadyToCommunicate* to activate global potential assignment.

start <***BeReadyToCommunicate***>
for each variable v_i $(i = 1, ..., |I|)$, *such that* $v \notin Assigned$, *do*
if $v_i \in Suspended$,
assign v_i*'s conditional probability locally;*
remove v_i *from Suspended;*
else
set $Rn = Rank$;
set $Objector = \emptyset$;
set $Pr = |\pi(v_i)|$;
set $Ph = \begin{cases} true: \text{ if } v_i \text{ is a physical local variable.} \\ false: \text{ otherwise.} \end{cases}$;
for each adjacent agent A_j $(j = 1, ..., n)$, *except* A_c, *do*
send $\{v_i, Pr, Ph$ *and* $Rn\}$ *to* A_j *and call it to run* ***UniformV***;
if A_j *sent back an objection,*
add A_j *to Objector;*
receive new Pr, Ph *and* Rn *from* A_j;
endif;
next A_j;
if $Objector = \emptyset$,
assign v_i*'s conditional probability locally;*
else
assign a uniform for v_i *locally;*
for each objector O_j $(j = 1, ..., |Objector| - 1)$, *do*

send v_i *to agent* O_j *and call it to run* ***WithdrawV****;*
next O_j*;*
endif;
endif;
add v_i *to Assigned;*
next v_i*;*
for each adjacent agent A_i $(i = 1, ..., n)$*, except* A_c*, do*
call A_i *to run* ***BeReadyToCommunicate****;*
next A_i*;*
end <***BeReadyToCommunicate***>

Algorithm *UniformV*

UniformV algorithm is responsible for assigning variable v a uniform belief locally when the agent has no objection available. It receives a variable v, its maximum number of parents Pr, its physical attribute Ph and the rank of the original requester agent Rn, from A_c and checks locally if A_* is more entitled to assign its local belief over v. If so, A_* assigns its local properties about v instead of the received ones. It in turn calls its adjacent agents to assign a uniform belief over v. In case of any adjacent agent has an objection, A_* withdraws its objection, and assigns a uniform belief over v. Finally, if any objection exists, no matter its source, A_* sends back an objection to A_c attached with the best properties about v.

start <***UniformV***>
receive $\{v, Pr, Ph$ *and* $Rn\}$ *from* A_c*;*
set $Objection = false$*;*
if $\big(|\pi(v)| > Pr$ *OR*
$(|\pi(v)| = Pr$ *AND* $Ph = false$ *AND* v *is a physical local variable) OR*
$(|\pi(v)| = Pr$ *AND* $Ph = v's$ *physical attribute AND* $Rank < Rn)\big)$*,*
update $Pr = |\pi(v)|$
update $Ph = \begin{cases} \text{true: } if\ v \text{ is a physical local variable.} \\ \text{false: } otherwise. \end{cases}$;
update $Rn = Rank$*;*
set $Objection = true$*;*
endif;
set $Objectors = \emptyset$*;*
for each adjacent agent A_i $(i = 1, ..., |adjacents|)$ *except* A_c*, such that* $v \in I_i$*, do*
send $\{v, Pr, Ph$ *and* $Rn\}$ *to* A_i *and call it to run* ***UniformV****;*
if A_i *sent back an objection,*
receive new $\{v, Pr, Ph$ *and* $Rn\}$ *from* A_i*;*
add A_i *to Objector;*
endif;
next A_i*;*
if $Objection = false$ *or* $Objector \neq \emptyset$*,*
assign a uniform for v *locally;*

add v to Assigned;
else,
add v to Suspended;
endif;
for each objector O_i $(i = 1, ..., |Objector| - 1)$, do
send v to agent O_i and call it to run ***WithdrawV****;*
next O_i;
if Objection = true or Objector $\neq \emptyset$,
send back an objection to A_c as well as the new $\{v, Pr, Ph$ and $Rn\}$;
else
send back an acceptance;
endif;
end < ***UniformV****>*

Algorithm *WithdrawV*

In *WithdrawV* algorithm, as a response to the call from A_c, agent A_* cancels its objection about the variable v in case of having an objection and assigns it a uniform belief locally. If it does not have an objection, and because it still keeps the objector set, it calls only the last objector in the set to withdraw its objection and assign v a uniform belief.

start < ***WithdrawV****>*
receive v from A_c;
if $v \in$ Suspended,
assign a uniform for v locally;
add v to Assigned;
remove v from Suspended;
else
send v to agent $Objector_{|Objector|}$ and call it to run ***WithdrawV****;*
endif;
end < ***WithdrawV****>*

Example

Here is an example to show how the proposed method can correctly assign the belief of the d-sepnodes in an LJF. Figure 4.3 shows a hypertree consisting of four agents and their subgraphs, S_0, S_1, S_2, and S_3 with belief assignment from this method labeled. For simplicity, we hid the non-d-sepnodes' belief assignment and we suppose all variables in the system are virtual variables (their physical attribute is $false$).

We start by giving each agent (A_0, ..., A_3) a rank value, let say 3, 1, 2, and 0 respectively. By calling A_0 to run **BeReadyToCommunicate**, the following processes will be done:

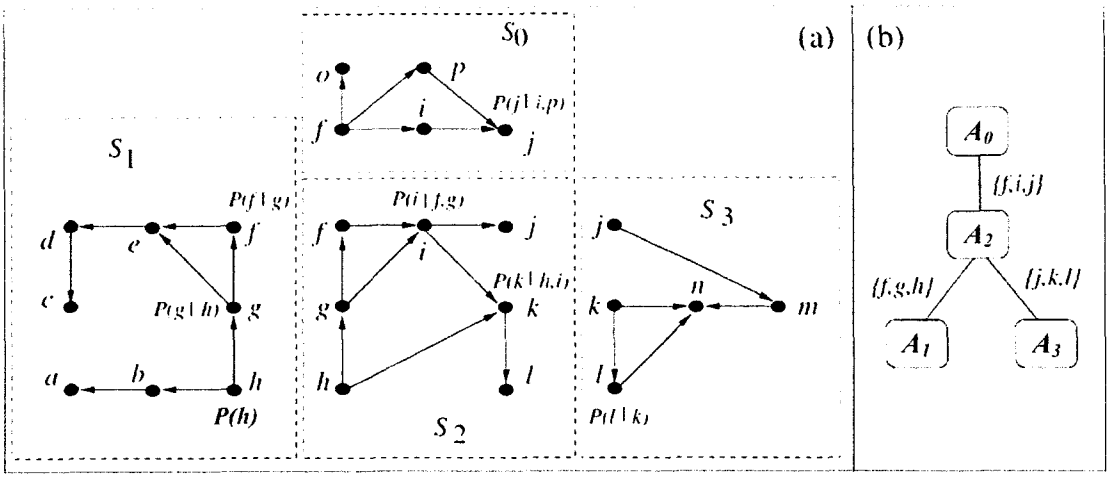


Figure 4.3: A simple MSBN.(a) Subnets. (b) The hypertree.

A_0 calls A_2 to run **UniformV**$(f, 0, false, 3)$.
A_2 calls A_1 to run **UniformV**$(f, 1, false, 2)$.
A_1 adds f to *Suspended* set and sends back an objection to A_2 with $(1, false, 1)$.
A_2 assigns a uniform belief to f locally and adds f to *Assigned* set.
A_2 sends back an objection to A_0 with $(1, false, 1)$.
A_0 assigns a uniform belief to f locally and adds f to *Assigned* set.
A_0 calls A_2 to run **UniformV**$(i, 1, false, 3)$.
A_2 adds i to *Suspended* set and sends back an objection to A_0 with $(2, false, 2)$.
A_0 assigns a uniform belief to i locally and adds i to *Assigned* set.
A_0 calls A_2 to run **UniformV**$(j, 2, false, 3)$.
A_2 calls A_3 to run **UniformV**$(j, 2, false, 3)$.
A_3 assigns a uniform belief to j locally, adds j to *Assigned* set and sends back an acceptance to A_2.
A_2 assigns a uniform belief to j locally, adds j to *Assigned* set and sends back an acceptance to A_0.
A_0 assigns j's conditional probability locally and adds j to *Assigned* set.
A_0 calls A_2 to run **BeReadyToCommunicate**.
A_2 assigns i's conditional probability locally, removes i from *Suspended* set and adds i to *Assigned* set.
A_2 calls A_1 to run **UniformV**$(g, 1, false, 2)$.
A_1 adds g to *Suspended* set and sends back an objection to A_2 with $(1, false, 1)$.
A_2 assigns a uniform belief to g locally and adds g to *Assigned* set.
A_2 calls A_1 to run **UniformV**$(h, 0, false, 2)$.
A_1 adds h to *Suspended* set and sends back an objection to A_2 with $(0, false, 1)$.
A_2 assigns a uniform belief to h locally and adds h to *Assigned* set.
A_2 calls A_3 to run **UniformV**$(k, 2, false, 2)$.
A_3 assigns a uniform belief to k locally, adds k to *Assigned* set and sends back an acceptance to A_2.
A_2 assigns k's conditional probability locally and adds k to *Assigned* set.
A_2 calls A_3 to run **UniformV**$(l, 1, false, 2)$.
A_3 adds l to *Suspended* set and sends back an objection to A_2 with $(1, false, 0)$.
A_2 assigns a uniform belief to l locally and adds l to *Assigned* set.
A_2 calls A_1 to run **BeReadyToCommunicate**.
A_1 assigns f's conditional probability locally, removes f from *Suspended* set and adds f to *Assigned* set.
A_1 assigns g's conditional probability locally, removes g from *Suspended* set and adds g to *Assigned* set.
A_1 assigns h's conditional probability locally, removes h from *Suspended* set and adds h to *Assigned* set.
A_2 calls A_3 to run **BeReadyToCommunicate**.
A_3 assigns l's conditional probability locally, removes l from *Suspended* set and adds l to *Assigned* set.

From the preceding progress, $P(j|i,p)$ is assigned in A_0 because no other agent has local $|\pi(j)|$ more than A_0. For $P(f|g)$, $P(g|h)$ and $P(h|\emptyset)$, both agents A_1 and

A_2 have the maximum value of $|\pi(x)|$ but because A_1 has better rank, it assigned them locally. The maximum value of $|\pi(x)|$ for both i and k are found in A_2, so it assigned $P(i|f,g)$ and $P(k|h,i)$ locally. For $P(l|k)$, both agents A_2 and A_3 have the maximum value of $|\pi(l)|$ but because A_3 has better rank, it assigned $P(l|k)$ locally. All leaf agents (except the one called by the system coordinator) in an LJF in their turns of running **BeReadyToCommunicate** do not need to send any e-message asking for acceptance because all their variables are already assigned or suspended. Thus, each leaf agent just assigns its conditional probabilities of its suspended variables locally.

Now we can clearly conclude that this proposal algorithm is efficient and adequate to satisfy Xiang's 5^{th} assumption in case of a real distributed multi-agent reasoning system without revealing any agent's internal knowledge.

Chapter 5

Artificial Neural Networks Playing a Role in Multi-agent Reasoning Systems

5.1 Introduction to Neural Networks

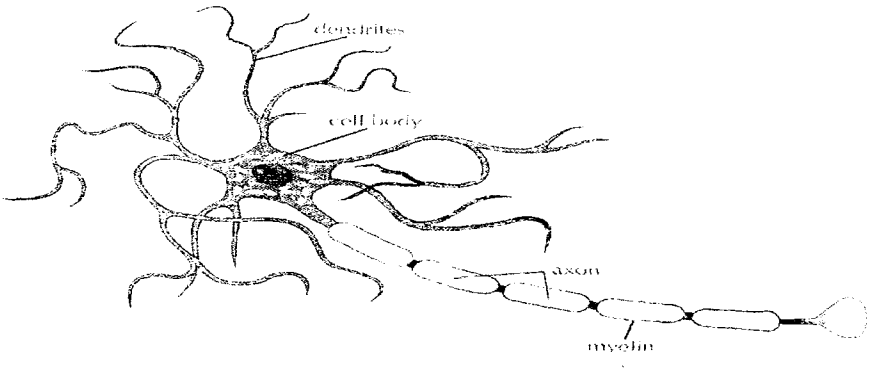


Figure 5.1: A biological neuron.

How a face in a crowd can be recognized? How the weather condition can be predicted by a forecaster or a weather analyst? Faced such problems, the human brain uses a network of interconnected processing elements called *neurons* to process information. Each neuron is autonomous and independent. It does its work asynchronously. The two problems posed, namely recognizing a face and forecasting the weather condition, have two important characteristics that distinguish them from other problems: First, the problems are complex, that is, you can not create a simple step-by-step algorithm or accurate formula to give an answer. Second, the available data to resolve the problems is equally complex and may be noisy or incomplete. The huge processing power inherent in biological neural structures has inspired the study of the structure itself for hints on organizing human-made

computing structures. *Artificial neural networks (ANN)* cover the way to organize synthetic neurons to solve the same kind of difficult, complex problems in a similar manner as we think the human brain may.

Since late 1980s, there has been an explosion in research of ANNs. Today, ANNs successful applications are reported across a big range of fields. ANN is a paradigm of learning tool, which is able to discover underlying dependencies between the given inputs and outputs by using training data sets. After the training process, it represents high-dimensional nonlinear functions. Many research institutions, industries, and commercial firms have already started to apply ANN successfully to many diverse types of real world problems. The most important applications include the following, [13]:

- Classification and pattern recognition for visual, sound, olfactory and tactile patterns.
- Time series forecasting for financial, weather, engineering time series.
- Diagnostics, e.g., in medicine or engineering.
- Robotics, including control, navigation, coordination, object recognition problems.
- Process control, like nonlinear and multivariate control of chemical plants, power stations and vehicles or missiles.
- Optimization, such as combinatorial problems, e.g., resource scheduling and routing.
- Signal processing, speech and word recognition.
- Machine vision, e.g., inspection in manufacturing, check reader, face recognition and target recognition.
- Financial forecasting for interest rates and stock indices, currencies.
- Financial services, like credit worthiness, forecasting and data mining, services for trade like segmentation of customer data.

An ANN function differs based on its application, e.g. In certain application areas, such as speech and word recognition, neural networks outperform conventional statistical methods. While in other fields, such as specific areas in robotics and financial services, they show promising application in real world situations. One of the first successful applications was a project (Sejnowski and Rosenberg 1987), aimed at training an ANN to pronounce English text consisting of seven consecutive characters from written text, presented in a moving window that gradually

scanned the text. The nonlinear nature of ANNs, the ability of neural networks to learn from their environments in supervised and unsupervised ways, as well as the universal approximation property of neural networks make them highly suited for solving difficult signal processing problems. For practical understanding of ANNs, it is imperative to develop a proper understanding of basic ANN structures and how they impact training algorithms and applications.

A challenge in surveying the field of ANN paradigms is to identify those ANN structures that have been successfully applied to solve real world problems from those that are still under development or have difficulty scaling up to solve realistic problems. It is also critical to understand the nature of the problem formulation so that the most appropriate ANN paradigm can be applied. In addition, it is also important to assess the impact of ANNs on the performance, robustness, and cost-effectiveness of the systems.

Learning is defined by Herbert Simon [20] as "any change in a system that allows it to perform better the second time on repetition of the same task or another task *drawn from the same population.*"

Artificial learning is then defined as the methodologies of modulating the operation of Learning and the ability of determining a method or a process to achieve learning.

Many methodologies were proposed to the artificial learning subject, [15]:

- ***Symbol-based learning***:A symbol-based learning method uses a set of symbols that represent the entities and relationships of a problem domain. Symbolic learning algorithms attempt to infer novel, valid and useful generalizations that can be expressed using these symbols.

- ***Connectionist learning***: The connectionist approaches represent knowledge as patterns of activity in networks of small, individual processing units. Inspired by the architecture of animal brains, connectionist networks learn by modified their structure and weights in response to training data. Rather than searching through the possible generalizations afforded by a symbolic representation language, connectionist models recognize invariant patterns in data and represent these patterns within their own structure.

- ***Emergent Learning***: Just as connectionist networks are inspired by the biological neural system, the emergent models are inspired by genetic or evolutionary analogs. The learning through genetic algorithms reflects the old concept the learning of human and animal systems that have evolved towards equilibrium with world.

5.1.1 Artificial Neural Networks

An *artificial neural network ANN*, also known as a *parallel distributed processing network*, is a computational structure that is inspired by the study of biological

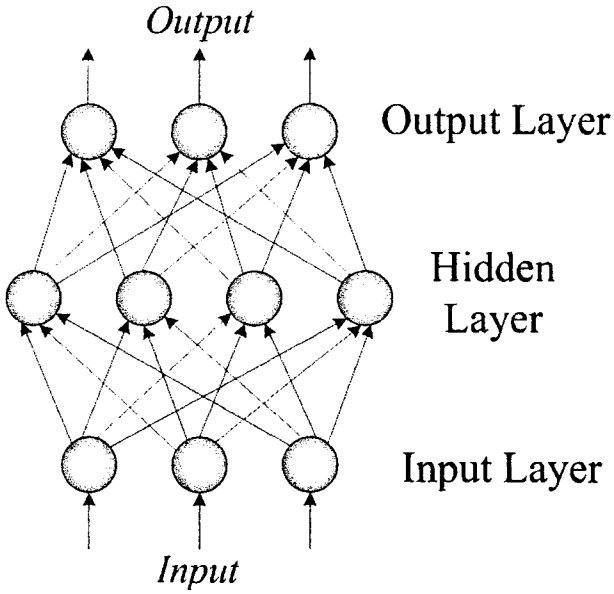


Figure 5.2: A typical layered feed-forward ANN.

neural processing. It consists of interconnected *processing elements* called *nodes* or *neurons* connected through weighted links called *connections*. Those neurons work together to produce an output function. So the output of an ANN relies on the cooperation of the individual neurons within the network to operate. Processing of information by ANNs is characteristically done in *parallel* rather than in *series* (or *sequentially*)

A *layered feed-forward ANN* has layers of processing elements. A layer of processing elements makes independent computations on data that it receives and passes the results to another layer. The next layer may in turn make its independent computations and pass on the results to yet another layer. Finally, a subgroup of one or more processing elements determines the output from the network.

Each neuron makes its computation based upon a *weighted sum* of its inputs that comes to it through its input connections, generates one output, and then spread it over its output connections to be processed again by other connected neurons. These connections have some feature of changing the strength of the passing signal according to its connection weight. The output connection from a neuron can be an input to another neuron or a final output of the ANN. The input connection to a neuron can be an output of another neuron or an initial input to the ANN. The first layer is called the *input layer*, the last the *output layer*, and the layers that are placed between the first and the last layers are the *hidden layers*.

The processing that is accomplished at any neuron over its inputs is established through applying a specific function called *net function*. The output of this function

is the value of the neuron after processing its inputs. This value is called a *net value.* Another function called *threshold activation function* or *output function* is sometimes used to qualify the output of a neuron in the output layer.
Connections between neurons are represented by edges of a directed graph in which the nodes are the neurons. Figure 5.2 is a layered feed-forward ANN with three layers, an input layer, a hidden layer and an output layer. The ball-shaped nodes represent neurons. The directed links show the weighted connections between nodes from a given layer and other nodes in an adjacent layer.

In ANN processing begins with the entire network in a quiescent state, an external comprised set of signals to be processed by the network is applied to the input layer. Each neuron then generates a single output signal with a magnitude that is a function of the total simulations received by it. Collectively, the output produced by all processing elements on the layer are then passed as input pattern to the subsequent layer, until the output layer produces output for the current input pattern [8].

5.1.2 Basic ANN Components

The Neuron

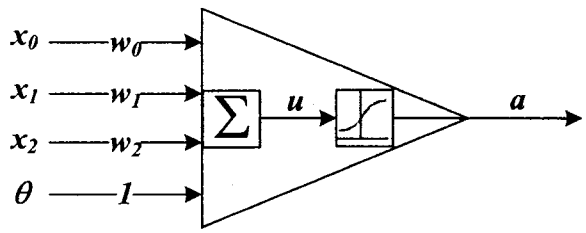


Figure 5.3: McCulloch-Pitts neuron model.

Among numerous ANN models that have been proposed over the years, all share the neuron as a common building block for its networked interconnected structures. The most widely used neuron model is *McCulloch-Pitts neuron* model illustrated in Figure 5.3 [19]. In Figure 5.3, each neuron consists of two parts, the *net function* and the *activation function.* The net function determines how the network inputs $\{x_i : 0 \leq i < n\}$ (*where n is the number of input connection*) are combined inside the neuron. In this figure, a weighted linear combination is adopted:

$$u = \sum_{i=0}^{n-1} w_i * x_i + \theta \tag{5.1}$$

$\{x_i : 0 \leq i < n\}$ are parameters expressing the synaptic weights. The quantity θ is called the *bias* and is used to model the threshold. In the literature, other

types of network input combination methods have been proposed as well. They are summarized in Table 5.1.

NET FUNCTIONS	FORMULA
Linear	$u = \sum_{j=1}^{n} w_j x_j + \theta$
Higher order ($2^n d$ order formula exhibited)	$u = \sum_{j=1}^{N} \sum_{j=1}^{N} w_{jk} x_j x_k + \theta$
Delta ($\sum - \prod$)	$u = \prod_{j=1}^{N} w_j x_j$

Table 5.1: Summary of Net Functions

The output of the neuron, denoted by a in Figure 5.3, is related to the network input u via a linear or nonlinear transformation called the activation function:

$$a = f(u) \tag{5.2}$$

In various ANN models, different activation functions have been proposed. The most commonly used activation functions are summarized in Table 5.2, [8].

Activation Function	Formula	Derivatives $\frac{df(u)}{du}$
Sigmoid	$f(u) = \frac{1}{1+e^{-u/T}}$	$f(u)[1 - f(u)]/T$
Hyperbolic tangent	$f(u) = tanh(\frac{u}{T})$	$\left(1 - [f(u)]^2\right)/T$
Inverse tangent	$f(u) = \frac{2}{\pi} tan^{-1}(\frac{u}{T})$	$frac2\pi T.frac11 + (u/T)^2$
Threshold	$f(u) = \begin{cases} 1 & if\ u \geq threshold \\ -1 & otherwise \end{cases}$	No $\frac{df(u)}{du}$ at $u = threshold$
Gaussian radial basis	$f(u) = exp[-u\|\|u - m\|\|^2/\sigma^2]$	$-2(u - m)f(u)/\sigma^2$
Linear	$f(u) = au + b$	a

Table 5.2: Neuron Activation Functions

Table 5.2 lists both the activation functions as well as their derivatives. In both *sigmoid* and *hyperbolic tangent* activation functions, derivatives can be computed directly from the knowledge of $f(u)$.

ANN Topology

In an ANN, multiple processing elements are interconnected to form a network to facilitate distributed computing. The configuration of the interconnections can be described efficiently with a directed graph. A directed graph consists of nodes (in the case of a neural network, neurons, as well as external inputs) and directed

(b) Acyclic Topology *(b) Cyclic Topology*

Figure 5.4: Illustration of an acyclic graph (a) and a cyclic graph (b). The cycle in (b) is emphasized with thick lines.

arcs (in the case of a neural network, synaptic links). The topology of the graph can be categorized as either acyclic or cyclic. In Figure 5.4a, an ANN with acyclic topology consists of no feedback loops. Such an acyclic neural network is often used to approximate a nonlinear mapping between its inputs and outputs. Figure 5.4b shows an ANN with cyclic topology contains at least one cycle formed by directed arcs. Such an ANN is also known as a recurrent network. Due to the feedback loop, a recurrent network leads to a nonlinear dynamic system model that contains internal memory. Recurrent ANNs often exhibit complex behaviors and remain an active research topic in the field of ANNs.

5.1.3 Multi-layer perceptron (MLP) model

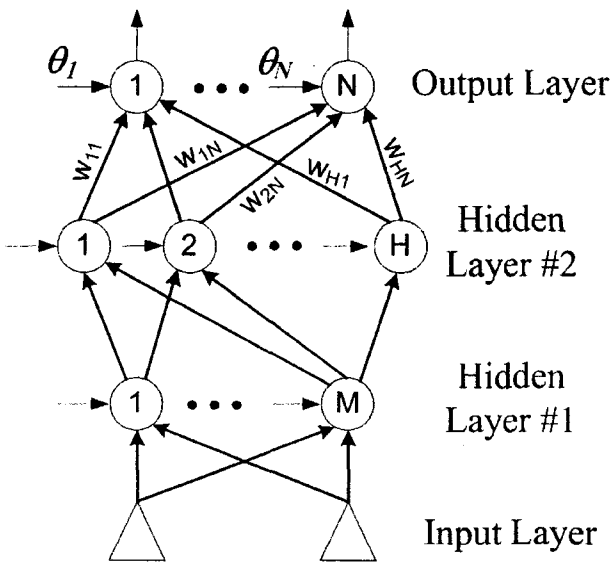


Figure 5.5: A three-layer multilayer perceptron configuration.

This is the most well known and most popular ANN among all the existing ANN paradigms. It consists of a feed-forward, layered network of neurons. Each neuron in an MLP has a nonlinear activation function that is often continuously differentiable. Some of the most frequently used activation functions for MLP include

the *sigmoid* function and the *hyperbolic tangent* function. Figure 5.5 illustrates a popular configuration of MLP where interconnections are provided only between neurons of successive layers in the network. In practice, any acyclic interconnections between neurons are allowed. Each ball-shape in the figure represents an individual neuron. These neurons are organized in layers, labeled as the hiddden layer #1, hidden layer #2, and the output layer. While the inputs at the bottom are also labeled as the input layer, there is usually no neuron model implemented in that layer. The name hidden layer refers to the fact that the output of these neurons will be feeded into upper layer neurons and, therefore, is hidden from the user who only observes the output of neurons at the output layer.

It has been proven that with a sufficient number of hidden neurons, an MLP with as few as two hidden layer neurons is capable of approximating an arbitrarily complex mapping within a finite support [2].

5.1.4 Error Back-Propagation Training of MLP

A key step in applying an MLP model is to choose the weight matrices. Assuming a layered MLP structure, the weights feeding into each layer of neurons form a weight matrix of that layer (the input layer is excluded as it contains no neurons). The values of these weights are found using the *error back-propagation training* method.

Finding the Weights of a Single Neuron MLP

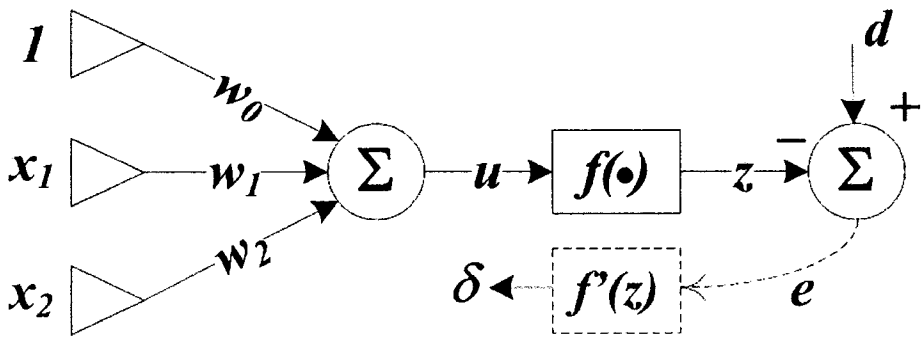


Figure 5.6: MLP example for back-propagation trainingsingle neuron case.

Let us first consider a simple example consisting of a single neuron to illustrate this procedure. Figure 5.6 represents the neuron in two separate parts: a summation unit to compute the net value u, and a nonlinear activation function to computer the neuron's output $z = f(u)$. Then the output z is to be compared with a desired target value d, and their difference, the error $e = d - z$, will be computed. There are two inputs $[x_l \ \ x_2]$ with corresponding weights w_1 and w_2. The input labeled with a constant 1 represents the bias term θ. Here, the bias link weight is labeled w_0. The net value is computed as:

$$u = \sum_{i=0}^{2} w_i x_i = Wx \tag{5.3}$$

where $x_0 = 1$, $W = [w_0\ w_1\ w_2]$ is the weight matrix, and $x = [1\ x_1\ x_2]^T$ is the input vector.

Given a set of training samples $\{(x(k), d(k)) : 1 \leq k \leq K\}$, the error back-propagation training begins by feeding all K inputs through the MLP network and computing the corresponding output $\{z(k) : 1 \leq k \leq K\}$. We usually use an initial random values for the weight matrix W. Then a sum of square error will be computed as:

$$\begin{aligned} E &= \sum_{k=1}^{K} [e(k)]^2 \\ &= \sum_{k=1}^{K} [d(k) - z(k)]^2 \\ &= \sum_{k=1}^{K} [d(k) - f(Wx(k))]^2 \end{aligned} \tag{5.4}$$

Now, the objective is to adjust the weight matrix W to minimize the error E. This leads to a nonlinear least square optimization problem. There are numerous nonlinear optimization algorithms available to solve this problem. Basically, these algorithms adopt a similar iterative formulation:

$$W(t+1) = W(t) + \Delta W(t) \tag{5.5}$$

where $\Delta W(t)$ is the correction made to the current weights $W(t)$.

Because different algorithms differ in the form of $\Delta W(t)$, we focus on a method called *the steepest descend gradient method.* It is the basis of the error back-propagation learning algorithm. The derivative of the scalar quantity E with respect to individual weights can be computed as follows:

$$\begin{aligned} \Delta W(t) &= -\eta\ g(t) \\ &= -\eta\ \frac{\partial E}{\partial W} \end{aligned} \tag{5.6}$$

where g is known as the gradient vector. η is called the *step size*, *learning rate* or *learning factor.* Usually, it is a value between 0 and 1, and specified by the network designer.

The derivative of the scalar quantity E with respect to individual weights can be computed as follows:

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \sum_{k=1}^{K} \frac{\partial [e(k)]^2}{\partial w_i} \\ &= \sum_{k=1}^{K} 2[d(k) - z(k)]\Big(-\frac{\partial z(k)}{\partial w_i}\Big), \quad for\ i = 0, 1, 2 \end{aligned} \tag{5.7}$$

where

$$\begin{aligned}\frac{\partial z(k)}{\partial w_i} &= \frac{\partial f(u)}{\partial u}\frac{\partial u}{\partial w_i} \\ &= f'(u)\frac{\partial}{\partial w_i}\left(\sum_{j=0}^{2} w_j x_j\right) \\ &= f'(u)x_i \end{aligned} \tag{5.8}$$

Hence,

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{K}[d(k) - z(k)]f'\big(u(k)\big)x_i(k) \tag{5.9}$$

With $\delta(k) = [d(k) - z(k)]f'\big(u(k)\big)$, the above equation can be expressed as:

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{K}\delta(k)x_i(k) \tag{5.10}$$

$\delta(k)$ is the error that represents the amount of correction needed to be applied to the weight w_i for the given input $x_i(k)$. The overall change Δw_i is thus the sum of such contribution over all K training samples. Therefore, the weight update formula has the format of:

$$w_i(t+1) = w_i(t) + \eta\sum_{k=1}^{K}\delta(k)x_i(k) \tag{5.11}$$

If a sigmoid activation function $f(u) = \frac{1}{1+e^{-u/T}}$, as defined in Table 5.2, is used, then the derivative $f'(u)$ is

$$f'(u) = f(u)\cdot\big[1 - f(u)\big] \tag{5.12}$$

and $\delta(k)$ can be computed as:

$$\delta(k) = \frac{\partial E}{\partial u} = \big[d(k) - z(k)\big]\cdot z(k)\cdot\big[1 - z(k)\big] \tag{5.13}$$

So far, we discussed how to adjust the weights of an MLP with a single layer of neurons.

Error Back-Propagation in a Multiple Layer Perceptron

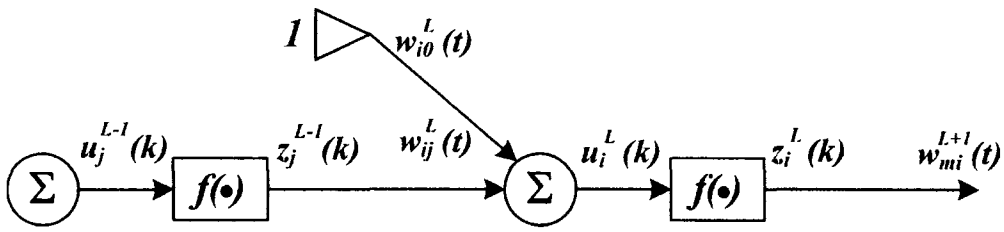


Figure 5.7: Notations used in a multiple-layer MLP model.

Now we discuss how to perform training for a multiple layer MLP of neurons. First, some new notations are adopted to distinguish neurons at different layers. In Figure 5.7, the net-function and output corresponding to the k^{th} training sample of the j^{th} neuron of the $(L-1)^{th}$ are denoted by $u_j^{L-1}(k)$ and $z_j^{L-1}(k)$, respectively. The input layer is the $zero^{th}$ layer. In particular, $z_j^0(k) = x_j(k)$. The output is feeded into the i^th neuron of the L^th layer via a synaptic weight denoted by $w_{ij}^L(t)$ or, for simplicity, w_{ij}^L, since we are concerned with the weight update formulation within a single training epoch.
To derive the weight adaptation equation, $\frac{\partial E}{\partial w_{ij}^L}$ must be computed:

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}^L} &= -2\sum_{k=1}^{K} \frac{\partial E}{\partial u_i^L(k)} \cdot \frac{\partial u_i^L(k)}{\partial w_{ij}^L} \\
&= -2\sum_{k=1}^{K} \left[\delta_i^L(k) \cdot \frac{\partial}{\partial w_{ij}^L} \sum_{m=1}^{M} w_{im}^L z_m^{L-1}(k) \right] \\
&= -2\sum_{k=1}^{K} \delta_i^L(k) \cdot z_j^{L-1}(k) \qquad (5.14)
\end{aligned}$$

where $1 \leq m \leq M$, and M is the number of neurons in layer $(L-1)$.

In Equation 5.14, the output $z_j^{L-1}(k)$ can be evaluated by applying the k^th training sample $x(k)$ to the MLP with weights fixed to w_{ij}^L. However, the delta error term $\delta_i^L(k)$ is not readily available and has to be computed.

Recall that the delta error is defined as $\delta_i^L(k) = \frac{\partial E}{\partial u_i^L(k)}$. Figure 5.8 is now used to illustrate how to iteratively compute $\delta_i^L(k)$ from $\delta_m^{L+1}(k)$ and weights of the $(L+1)^th$ layer.

Note that $z_i^L(k)$ is fed into all M neurons in the $(L+1)^th$ layer. Hence:

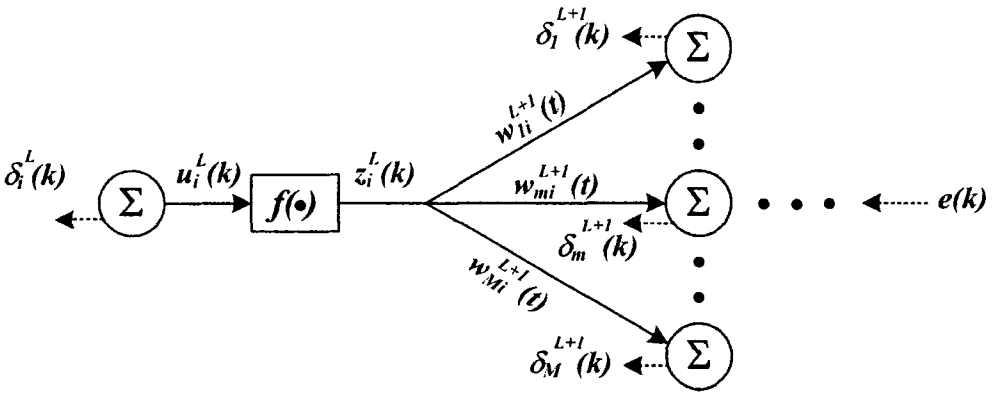


Figure 5.8: Illustration of error back-propagation computation.

$$\begin{aligned}
\delta_i^L(k) &= \frac{\partial E}{\partial u_i^L(k)} = \sum_{m=1}^{M} \frac{\partial E}{\partial u_m^{L+1}(k)} \cdot \frac{\partial u_m^{L+1}(k)}{\partial u_i^L(k)} \\
&= \sum_{m=1}^{M} \left[\delta_m^{L+1}(k) \cdot \frac{\partial}{\partial u_i^L(k)} \cdot \sum_{j=1}^{J} w_{mj}^L f\left(u_j^L(k)\right) \right] \\
&= f'\left(u_j^L(k)\right) \cdot \sum_{m=1}^{M} \delta_m^{L+1}(k) \cdot w_{mi}^L \qquad (5.15)
\end{aligned}$$

Equation 5.15 is the error back-propagation formula that computes the delta error from the output layer back toward the input layer, in a layer-by-layer manner.

5.2 Speeding up a Multi-agent Slow Inference by Employing ANNs

Although **CommunicateBelief** algorithm can be performed effectively, when the problem domain is very large, the computation of **CommunicateBelief** can still be *quite expensive*. Each agent must pass i-messages *twice* locally during **UnifyBelief** algorithm. To pass e-messages among agents, **AbsorbThroughLinkage** algorithm must be performed, which involves transmission across some media, ***resulting in delay***. Furthermore, the e-message passing and local i-message passing must be performed in a *semiparallel* fashion [21]. The *semiparallel* computation implies that the computational time of **CommunicateBelief** is lower bounded by *four times* the length of the longest hyperchain in the LJF.

Unlike inference computations in an agent in a hypertree which use explicit, often logical, rules arranged to manipulate believes in a serial manner (for each chain), however, ANN systems rely on parallel processing of data elements, using statistical properties instead of logical rules to transform information.

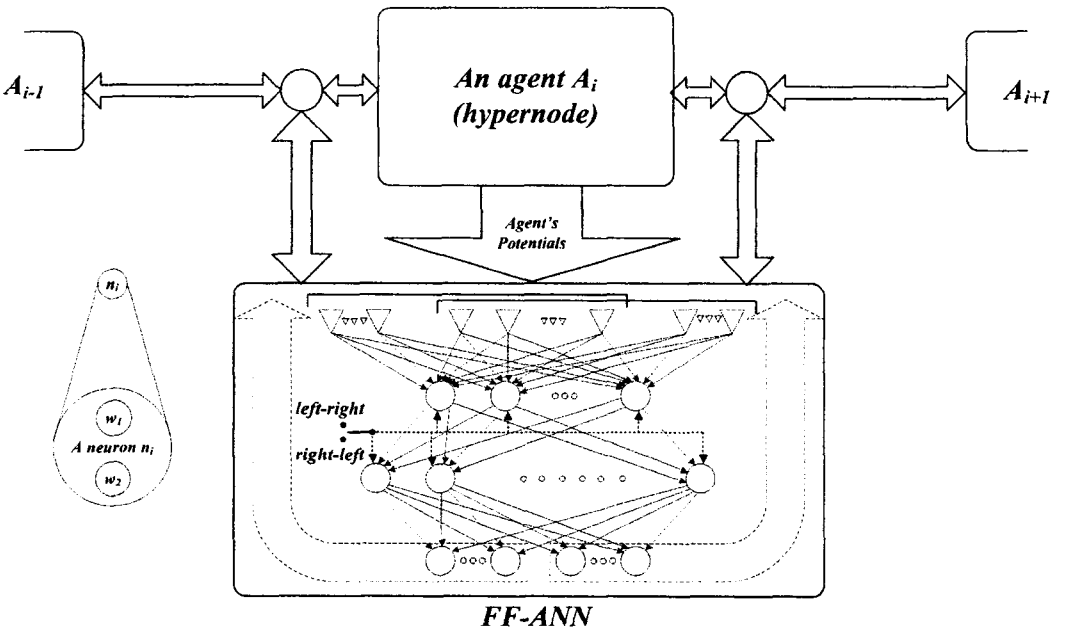


Figure 5.9: A feed-forward ANN, with dual-weight neurons in its hidden layers, associated with a hypernode.

ANNs process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements (neurons) working in parallel to solve a specific problem. ANNs' ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast response and computational times which are due to their parallel architecture. Hence, special hardware devices are being designed and manufactured to take advantage of this capability.

In an ANN, it is only possible to handle independent parts in parallel processes. That means only neurons belonging to the same layer can be run in parallel. For example, any neuron of the second hidden layer needs the outputs of the first hidden layer but not from other neurons within its own layer.

Given an ANN with single hidden layer needs only two processing stages to transform an input vector to its corresponding output vector. In contrast, having a trivial BN represented as a JT with a chain of just five clusters, see Figure 5.10, needs sixteen processing stages to perform **UnifyBelief**. Eight for each half (**CollectEvidence**, **DistributeEvidence**) and two for each adjacent pair of clusters.

Taking this as a starting point, we proposed a kind of cooperation between MSBNs and ANNs aiming to estimate e-messages in advance and speed up the inference and belief updating between agents among the LJF. The model can be adopted in

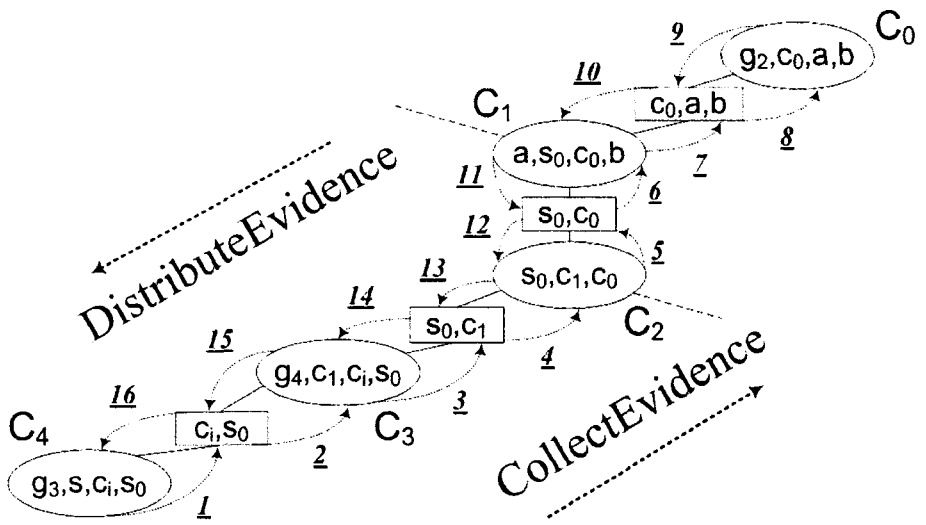


Figure 5.10: Illustration of UnifyBelief on a five-cluster chain in a JT.

the case of time-sensitive distributed multi-agent reasoning systems. Such time-sensitive systems in most cases have some tolerance. These systems can sacrifice little accuracy for ensuring satisfiable speed. The model is represented by the use of ANNs for prediction of e-messages that represent the belief over a d-sepset and that is by associating an ANN with each agent in parallel. Figure 5.9 gives a clear picture about our model. The ANN [1]captures the incoming e-message over one of the d-sepsets of its agent (left port or right port), [2]directs the transformation process to adapt to one of the two directions ($left-right$ or $right-left$), and [3]immediately generates the predicted e-message and sends it to the target agent while the local agent is busy by running **UnifyBelief**. We use the terms $right-left$, $left-right$ and the virtual switch on the figure just for simplification.

An ANN can learn the behavior of its agent by catching the input e-messages and the output e-messages and use them as a training sample and train itself using the error back-propagation method.

In our model, we propose the use of an ANN with dual-weight neurons in order to use only one ANN for each agent to do e-message prediction in both directions. We have to mention here that our model can be applied only on each agent that has exactly two adjacent agents making a chain with them.

Once an ANN is trained enough to do its responsibilities, its agent has to stop sending e-messages that carry belief and devote itself to receiving e-messages and to doing local belief unification after each reception.

In fact, this model consists of two ANNs but only one is working at a time. In Figure 5.9, if an e-message came from agent A_{i-1} towards A_i, the message will be received by both the agent and the ANN. Because the message came from the left side, the virtual switch will activate those weights in the neurons belonging to

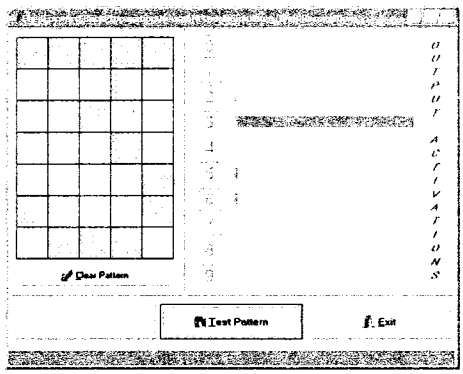


Figure 5.11: Feed forward ANN Recognizing the number "3".

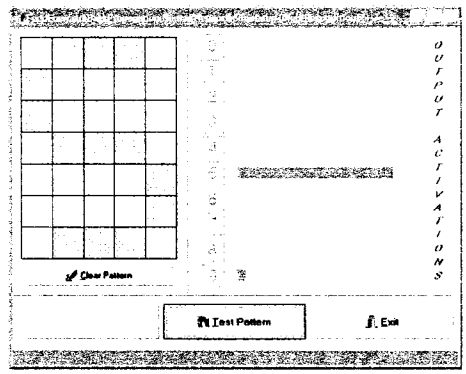


Figure 5.12: Feed forward ANN Recognizing the number "5".

the $left - right$ ANN. As a response the ANN processes the received e-message and generates its corresponding e-message from the other side. The inputs of the $left - right$ ANN are the e-message coming from A_{i-1} and the potentials of the associated agent A_i represented by the wide arrow from A_i towards the ANN. On the other hand, the inputs of the $right - left$ ANN are the e-message coming from A_{i+1} and the potentials of A_i. Therefore, the number of input lines and output neuron differ from one ANN to foanother in the same model. The figure shows clearly that $left - right$ and $right - left$ ANNs are sharing only the hidden layers.

A software program has been developed to implement this model and fortunately it gave very satisfiable results. The following section presents these results.

5.3 Implementation Results

5.3.1 Digit Recognition

First of all, we developed an object oriented C++ program with GUI to simulate a feed-forward ANN with a back-propagation learning algorithm. In general, the software deals with an ANN as one object. This object can be created dynamically and easily by passing the number or hidden layers, the number of neurons in each hidden layer, the size of the input vector and the size of output vector. In turn, an ANN as an object deals with each layer of neurons as a single object. It actually deals with only the input and output layers. Those layers, in turn, consist of smaller entities representing neurons. Each neuron receives commands from its parent layer and has the ability to communicate directly with the neurons in the preceding layer.

A standard feed-forward ANN was used in the program to recognize a digit given through an input pattern drawn using the mouse. It is designed to be trained using the training patterns and the back-propagation algorithm presented in Section 5.1.4 on page 61.

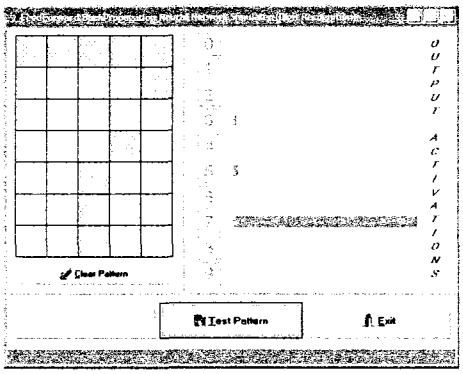


Figure 5.13: Feed forward ANN Recognizing the number "7".

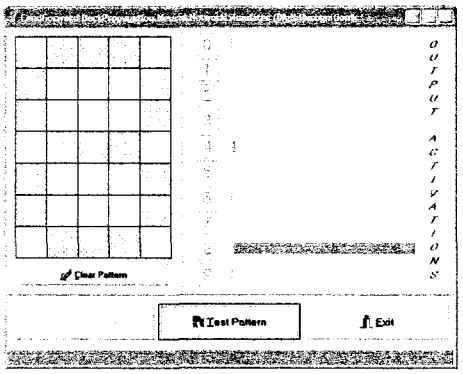


Figure 5.14: Feed forward ANN Recognizing the number "8".

Once the program is started, and before applying digits, a button labeled "Start Training" needs to be clicked to start training the network. During this operation a progress bar will be activated. After the training is finished, the training button automatically disabled and another button for testing patterns, "Start Training", will be enabled. When a pattern is plotted on the grid and the test button is clicked, the neural network processes the input pattern using its neurons' weights and then gives its output represented as an activation value for each digit from 0 to 9 and shaped as blue bars. Figures 5.11 through 5.13 illustrate the outputs of number of tests done.

5.3.2 E-message Prediction

One of the most challenging problems in training neural networks is determining the appropriate features that contribute the most to the training of the network. The same programming objects used in the preceding section are used here. They represent a standard feed-forward ANN and the error-backpropagation algorithm trained it using training sets taken from agent A_1 in the MSBN of the digital system shown in Figures 6.1 through 6.3. The hypertree is shown in Figure 5.15.

The network consisted of the input layer, one hidden layer and the output layer. The input layer consists of 296 inputs, one input for each potential value in a cluster inside the agent or in a linkage inside the received e-message. Variable number of hidden units are used and 96 output units, one unit per each potential value in the output e-message. The number of hidden units influences the training results of the network and ranged from 5 to 20 hidden units: 5, 10, 15 and 20. For these numbers, four networks were trained for a fixed size of the training set.

In each training cycle the same parameters were used. All the networks were trained for a maximum of 10,000 epochs with a momentum term of 0.9 and learning rate of 0.3. The training was stopped when the maximum number of epochs reached (10,000) or when the mean squared error dropped below 1.0e-5. We finally used 10 hidden units in the middle layer. The state of the network, i.e. the weights, were

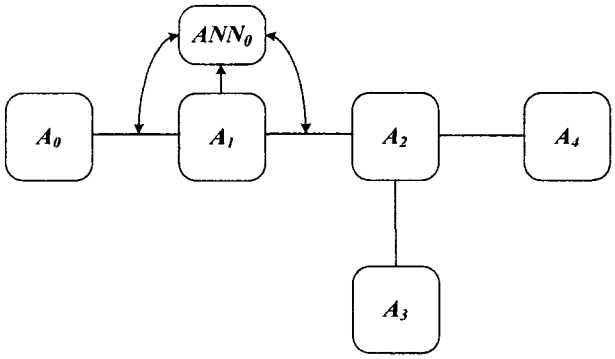


Figure 5.15: The hypertree of the digital system shown in Figures 6.1 through 6.3 with an ANN attached with agent A_1.

kept in a file.

Afterwards, the network was connected in parallel with agent A_1 as shown in Figure 5.15 and the object oriented framework, presented in the following chapter, was turned on. Two variables were considered instantiated and entered to the system one by one from different agents. Each time **CommunicateBelief** performed, our model generates an e-message consisting of 96 potential values and saves it in an output file.

In the following, results of two e-messages, and their corresponding charts, generated by the model and sent to A_2 are shown. For comparison, we attached each potential value generated by the ANN model (an actual value) with the corresponding one (a desired value) generated by agent A_1 in the form ($ANN's\ value$, $A_1's\ value$). The following e-message potential values are in the order of the linkages in the LT between A_1 and A_2 (for more details refer to "For the agent A_1:" → "Linkages with respect to "G2":" in Section 6.2.3 on page 90).

As it can be seen by the scale of vertical axis of the two Figures 5.16 and 5.17, the scale of differences between the desired and the actual e-messages are in the range of $0 - 0.0012$ in (b), which could not have been captured in (a). So in (b) in fact, the differences between the actual and the desired e-messages of the corresponding bars are compared. And as it is shown these differences are so small.

The results of e-message No(1):

==========

01.(0.000070, 0.000000),	02.(0.000045, 0.000000),	03.(0.000047, 0.000000),	04.(0.000060, 0.000000),
05.(0.020649, 0.020651),	06.(0.000076, 0.000062),	07.(0.002491, 0.002938),	08.(0.976280, 0.976349),
09.(0.000069, 0.000000),	10.(0.000049, 0.000000),	11.(0.617519, 0.617517),	12.(0.000066, 0.000000),
13.(0.000054, 0.000000),	14.(0.000041, 0.000000),	15.(0.000043, 0.000000),	16.(0.997372, 0.997879),
17.(0.000040, 0.000000),	18.(0.000045, 0.000000),	19.(0.001159, 0.001248),	20.(0.381232, 0.381236),
21.(0.000076, 0.000000),	22.(0.000039, 0.000000),	23.(0.001195, 0.000106),	24.(0.001658, 0.002016),
25.(0.993029, 0.992979),	26.(0.000061, 0.000000),	27.(0.000070, 0.000000),	28.(0.000055, 0.000000),
29.(0.000056, 0.000000),	30.(0.000052, 0.000000),	31.(0.000039, 0.000000),	32.(0.000043, 0.000000),
33.(0.000032, 0.000000),	34.(0.000027, 0.000000),	35.(0.000061, 0.000000),	36.(0.000068, 0.000000),
37.(0.992987, 0.992979),	38.(0.000077, 0.000000),	39.(0.000036, 0.000000),	40.(0.000050, 0.000000),
41.(0.006976, 0.007021),	42.(0.000044, 0.000000),	43.(0.000059, 0.000000),	44.(0.000038, 0.000000),

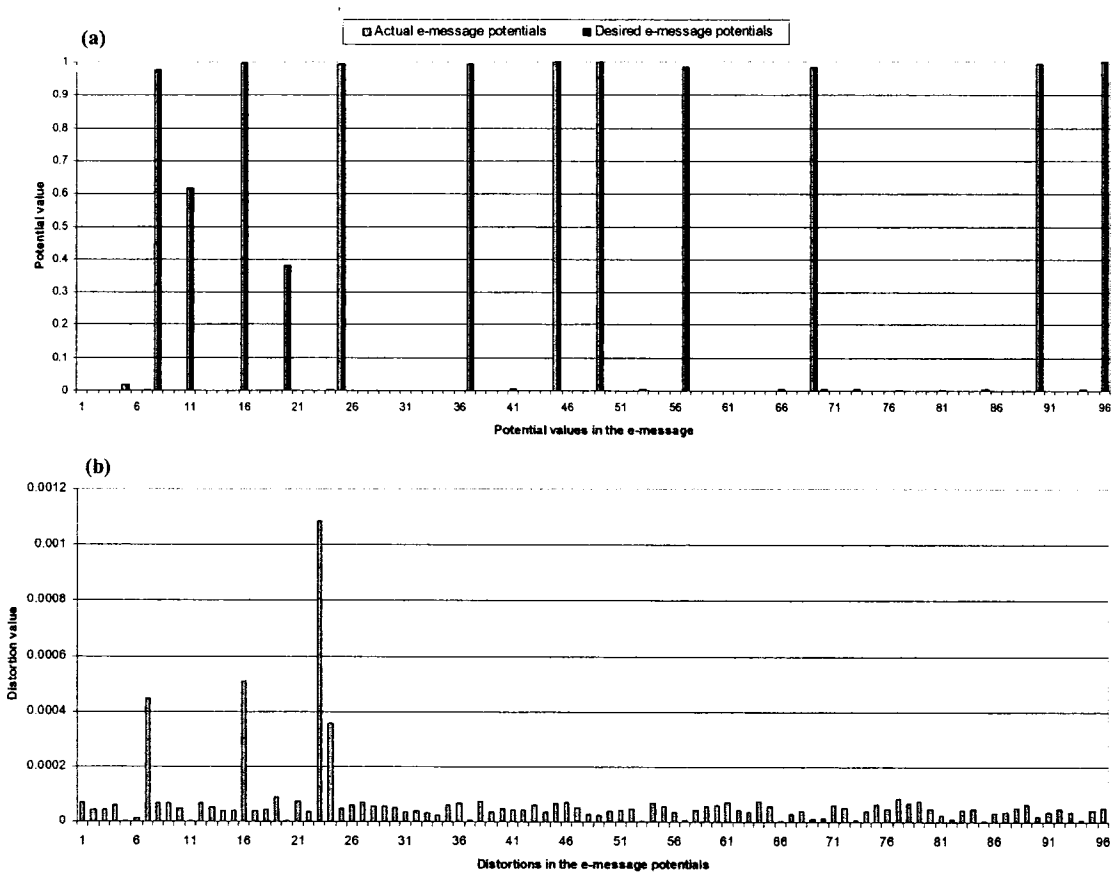


Figure 5.16: Two charts, the first shows the identity level between the actual output of $e - message_1$ (obtained from the proposed model) and the desired output (obtained from agent A_1). The second chart reflects how low the difference (distortion) between the actual $e - message_1$ and the desired $e - message_1$ is.

45.(0.999931, 1.000000), 46.(0.000070, 0.000000), 47.(0.000053, 0.000000), 48.(0.000031, 0.000000),
49.(0.999972, 1.000000), 50.(0.000041, 0.000000), 51.(0.000047, 0.000000), 52.(0.000050, 0.000000),
53.(0.007026, 0.007021), 54.(0.000067, 0.000000), 55.(0.000056, 0.000000), 56.(0.000039, 0.000000),
57.(0.985057, 0.985050), 58.(0.000045, 0.000000), 59.(0.000057, 0.000000), 60.(0.000060, 0.000000),
61.(0.000070, 0.000000), 62.(0.000046, 0.000000), 63.(0.000039, 0.000000), 64.(0.000077, 0.000000),
65.(0.000056, 0.000000), 66.(0.004954, 0.004950), 67.(0.000031, 0.000000), 68.(0.000041, 0.000000),
69.(0.985039, 0.985050), 70.(0.004934, 0.004950), 71.(0.000060, 0.000000), 72.(0.000051, 0.000000),
73.(0.006957, 0.006965), 74.(0.000057, 0.000015), 75.(0.000064, 0.000000), 76.(0.000050, 0.000000),
77.(0.002900, 0.002985), 78.(0.000082, 0.000015), 79.(0.000076, 0.000000), 80.(0.000050, 0.000000),
81.(0.002957, 0.002985), 82.(0.000048, 0.000035), 83.(0.000044, 0.000000), 84.(0.000048, 0.000000),
85.(0.006961, 0.006965), 86.(0.000067, 0.000035), 87.(0.000039, 0.000000), 88.(0.000052, 0.000000),
89.(0.000064, 0.000000), 90.(0.994999, 0.994975), 91.(0.000038, 0.000000), 92.(0.000050, 0.000000),
93.(0.000037, 0.000000), 94.(0.005033, 0.005025), 95.(0.000046, 0.000000), 96.(0.999946, 1.000000)

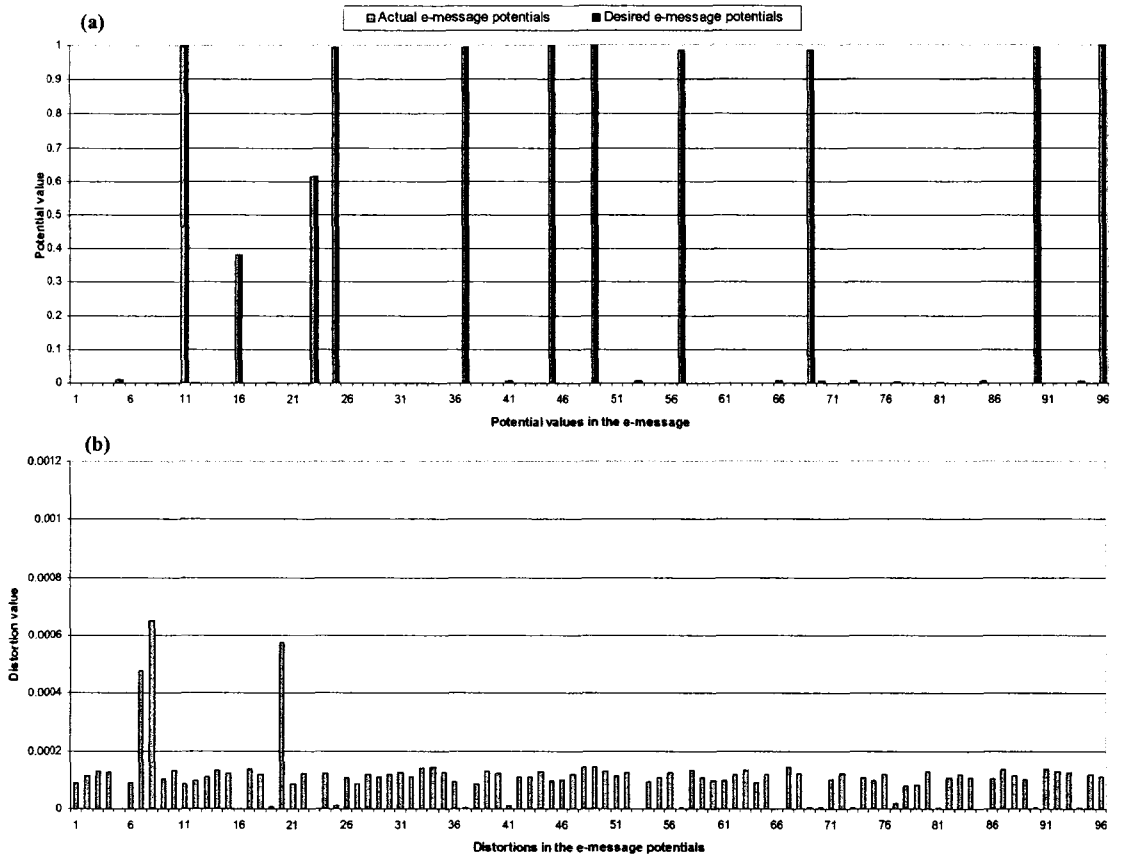


Figure 5.17: Two charts, the first shows the identity level between the actual output of $e - message_2$ (obtained from the proposed model) and the desired output (obtained from agent A_1). The second chart reflects how low the difference (distortion) between the actual $e - message_2$ and the desired $e - message_2$ is.

The results of e-message No(2):

==========

01.(0.000091, 0.000000),	02.(0.000116, 0.000000),	03.(0.000129, 0.000000),	04.(0.000125, 0.000000),
05.(0.007874, 0.007873),	06.(0.000115, 0.000024),	07.(0.000478, 0.000000),	08.(0.000754, 0.000103),
09.(0.000106, 0.000000),	10.(0.000132, 0.000000),	11.(0.997858, 0.997943),	12.(0.000096, 0.000000),
13.(0.000113, 0.000000),	14.(0.000132, 0.000000),	15.(0.000122, 0.000000),	16.(0.383128, 0.383127),
17.(0.000139, 0.000000),	18.(0.000121, 0.000000),	19.(0.002025, 0.002016),	20.(0.000614, 0.000041),
21.(0.000087, 0.000000),	22.(0.000122, 0.000000),	23.(0.616098, 0.616099),	24.(0.000897, 0.000774),
25.(0.992967, 0.992979),	26.(0.000109, 0.000000),	27.(0.000088, 0.000000),	28.(0.000118, 0.000000),
29.(0.000113, 0.000000),	30.(0.000118, 0.000000),	31.(0.000127, 0.000000),	32.(0.000113, 0.000000),

33.(0.000142, 0.000000), 34.(0.000146, 0.000000), 35.(0.000125, 0.000000), 36.(0.000095, 0.000000),
37.(0.992977, 0.992979), 38.(0.000087, 0.000000), 39.(0.000130, 0.000000), 40.(0.000123, 0.000000),
41.(0.007032, 0.007021), 42.(0.000113, 0.000000), 43.(0.000111, 0.000000), 44.(0.000131, 0.000000),
45.(0.999902, 1.000000), 46.(0.000102, 0.000000), 47.(0.000121, 0.000000), 48.(0.000144, 0.000000),
49.(0.999855, 1.000000), 50.(0.000131, 0.000000), 51.(0.000116, 0.000000), 52.(0.000125, 0.000000),
53.(0.007020, 0.007021), 54.(0.000094, 0.000000), 55.(0.000108, 0.000000), 56.(0.000125, 0.000000),
57.(0.985048, 0.985050), 58.(0.000132, 0.000000), 59.(0.000109, 0.000000), 60.(0.000098, 0.000000),
61.(0.000099, 0.000000), 62.(0.000121, 0.000000), 63.(0.000132, 0.000000), 64.(0.000092, 0.000000),
65.(0.000121, 0.000000), 66.(0.004949, 0.004950), 67.(0.000146, 0.000000), 68.(0.000123, 0.000000),
69.(0.985053, 0.985050), 70.(0.004953, 0.004950), 71.(0.000103, 0.000000), 72.(0.000122, 0.000000),
73.(0.006967, 0.006965), 74.(0.000124, 0.000015), 75.(0.000097, 0.000000), 76.(0.000118, 0.000000),
77.(0.003002, 0.002985), 78.(0.000096, 0.000015), 79.(0.000083, 0.000000), 80.(0.000129, 0.000000),
81.(0.002990, 0.002985), 82.(0.000142, 0.000035), 83.(0.000120, 0.000000), 84.(0.000109, 0.000000),
85.(0.006966, 0.006965), 86.(0.000139, 0.000035), 87.(0.000138, 0.000000), 88.(0.000117, 0.000000),
89.(0.000100, 0.000000), 90.(0.994970, 0.994975), 91.(0.000138, 0.000000), 92.(0.000131, 0.000000),
93.(0.000125, 0.000000), 94.(0.005023, 0.005025), 95.(0.000120, 0.000000), 96.(0.999887, 1.000000)

Chapter 6

Implementing a Multi-agent Reasoning System as an Object Oriented Framework

6.1 The Implementation Package

An object-oriented Bayesian network framework has been developed using the C++ programming language. This package has been given generality and is mainly designed to give an efficient and easy way to implement three kinds of BN representations: DAG, JT and LJF. It consists of three main C++ Object-Oriented classes: *BN_Tree*, *BN_Comp*, and *BN_JTree*. Each class inherits its preceding classes in a direct and indirect manner (e.g. *BN_Comp* directly inherits *BN_Tree*, and *BN_JTree* directly inherits *BN_Comp* and indirectly inherits *BN_Tree*). Each class consists of many classes representing the basic entities of the model it represents.

6.1.1 The Four Main O.O. Classes

The four classes are briefly described in the following:

1. The first class, *BN_Tree*, deals with a basic DAG and is responsible for loading data that defines and describes *qualitative* and *quantitative* components of the network, constructing the graph and performing the $\lambda-\pi$ message passing algorithm (if applicable). In the case of a multi-agent MSBN, it has a another duty and that is loading additional information regarding its subdomain's position among the subdomains of the given hypertree and the intersection sets with its adjacent agents.

2. The second class, *BN_Comp*, represents the compiler object that is responsible for performing the compilation procedure starting with moralization and

ending with generating the final clique graph. This class has the ability to play the same role in two situations. In a single-agent BN, it can be used to compile a given DAG and generates the corresponding moral, chordal, and clique graph. In a multi-agent MSBN, it can be used as an entity in each subdomain and controlled by its parent class, *BN_JTree*. In particular, it compiles a given subnet respecting all of its adjacent subdomains in a cooperative manner.

3. The third class, *BN_JTree*, is responsible for constructing the final form of the junction tree representation after obtaining the outputs of the associated *BN_Comp* object and applying Prim's algorithm. For a single agent BN, it assigns initial potentials and performs belief updating by using the absorption method. For a multi-agent MSBN, *BN_JTree* directs its compiler object to perform the local compilation stages as an entity in the hypertree, as well as doing belief updating as a response to the global communication algorithm, **CommunicateBelief**.

4. *BN_LTree* is the fourth class to handle the LJF structure. For each agent there is a single *BN_JTree* for each adjacent agent, and it is responsible for keeping the information about that adjacent agent including the physical address and the set of intersection variables, creating the linkage tree with respect to the associated agent's junction tree, and works as a communication channel or an e-message bridge between its parent, and the associated adjacent agent. Each *BN_JTree* object in a multi-agent MSBN has a *BN_LTree* object for each adjacent hypernode in the hypertree.

We have to mention here some of the package's advantages:

- Each object has its own members (properties, methods and operators), so it just needs to be activated or directed to perform its responsibilities by itself.
- Each object is provided with one or more constructors and destructors, so there is no need to tell it how or what kind of objects it has to initialize before usage or what members are to be destroyed before its destruction. It automatically and dynamically creates and initializes its important members.
- The package is designed to deal with different sizes of BNs. It allocates memory dynamically depending on the domain's description files given by the user. It has no limitations except the memory space available because all the indexing variables used are defined as *long int* inside the code.
- Each object has the ability to print out in a given file a short description about what it is doing at the moment, so the user can review the progress.
- In the case of a multi-agent LJF, all conversations between agents will be saved in a text file, so it can be explored or printed.

- Each class of the first three classes is defined in two separate source files: *.hpp* contains the prototype of the class and *.cpp* contains the code. Each *.hpp* file of *BN_Comp* and *BN_JTree* includes the *.hpp* file of its sub-class. So *BN_Tree* can be used by including only *DAG.hpp* and *DAG.cpp*, *BN_Comp* can be used by including only *DAG2JT.hpp* and *DAG2JT.cpp*, an *BN_JTree* by including only *JT.hpp* and *JT.cpp*.

6.1.2 The Structure of BN Description File

Before we show the implementation results, it is preferred to explain the structure of the input file that used to describe a BN or a subnet:

The following section gives the implementation results of applying an MSBN adopted from ([21] Figure 6.11).

6.2 Cooperative Multi-agent System Troubleshoots a Digital System

Here, we demonstrate how an MSBN-based multi-agent reasoning system functions in practice. We are using digital system monitoring as the problem domain. The digital system and its physical components are shown in Figures 6.2 and 6.1 respectively. We need five agents to populate the monitoring system. The virtual components are shown in Figure 6.3. The subnet dependence structures are shown in Figures 6.4 through 6.8. The hypertree is shown in Figure 3.1.

In addition to the dependence structures, we assume the following representational parameters:

- Each logical gate is represented as a binary variable and is either *normal* or $faulty$(the space is denoted as $\{good, bad\}$).
- It is assumed that each gate has a 0.01 probability of being faulty. A faulty gate is modeled so that it may or may not produce the incorrect output.
- A faulty *AND* gate is assumed to output correctly 20% of the time. A faulty *OR* gate outputs correctly 70% of the time, and a faulty *NOT* gate outputs correctly 50% of the time.

We consider the external inputs are as follows:

$a_0 = 0$, $a_2 = 1$, $b_0 = 0$, $e_2 = 1$, $h_2 = 0$, $i_0 = 1$, $i_2 = 1$, $k_0 = 0$, $k_1 = 0$, $l_2 = 0$, $o_0 = 0$, $o_1 = 0$, $o_2 = 1$, $p_1 = 1$, $s_0 = 0$, $v_4 = 1$, $v_7 = 1$, $y_1 = 1$, $y_2 = 1$, $z_1 = 0$, $z_5 = 0$.

First of all, we created five description files structured as in Tabel 6.1, one for each subnet dependence structure or subDAG. Five *JT_Tree* objects were used, each one representing one agent in the system. Each object has been provided by one of the description files, so it loaded the structure of its associated subnet

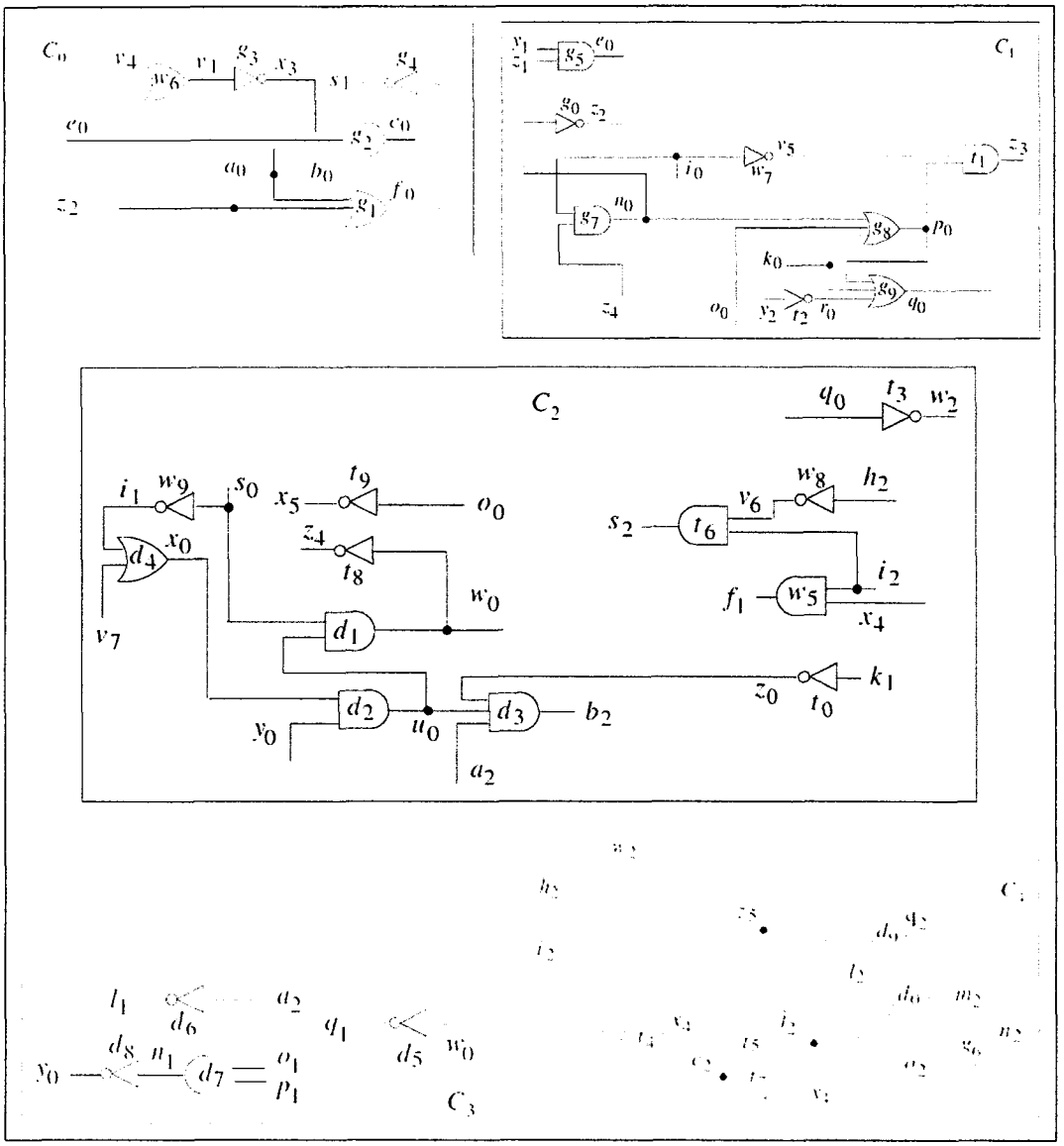


Figure 6.1: The five physical components of a digital system.

Title	Type	Example
Graph title	text	DAG_1
Total number of variables	integer	5
Number of roots	integer	1
Number of instantiated variables	integer	1
1^{st} variable		
Title	text	X
Number of states	integer	2
States	text	x_1
	text	x_2
Number of parents	integer	0
Parents (if applicable)	text	
CPT	real	0.1
	real	0.9
2^{nd} variable		
Title	text	Y
Number of states	integer	2
States	text	y_1
	text	y_2
Number of parents	integer	1
Parents (if applicable)	text	X
CPT	real	0.6
	real	0.2
	real	0.4
	real	0.8
⋮		
5^{th} variable		
Title	text	W
Number of states	integer	2
States	text	w_1
	text	w_2
Number of parents	integer	1
Parents (if applicable)	text	Y
CPT	real	0.9
	real	0.3
	real	0.1
	real	0.7
1^{st} instantiated variable		
Title	text	Y
Value	text	y_2
Number of adjacent graphs (if applicable)	integer	1
1^{st} adjacent graph title	text	DAG_2
Number of intersected variables	integer	2
1^{st} variable	text	Y
2^{nd} variable	text	W

Table 6.1: BN description file structure.

and became connected with its adjacent objects. At this moment, the multi-agent reasoning system is ready to perform the cooperative compilation.

6.2.1 Results of the cooperative global moralization

Here, according to **CoMoralize** algorithm (Section 3.4.2 on page 32), the system coordinator (represented by the main program) calls an object arbitrarily (implemented by *rand*() function) to run **CollectMLinks**. After it finishes the coordinator calls the same object to run **DistributeMLinks**.

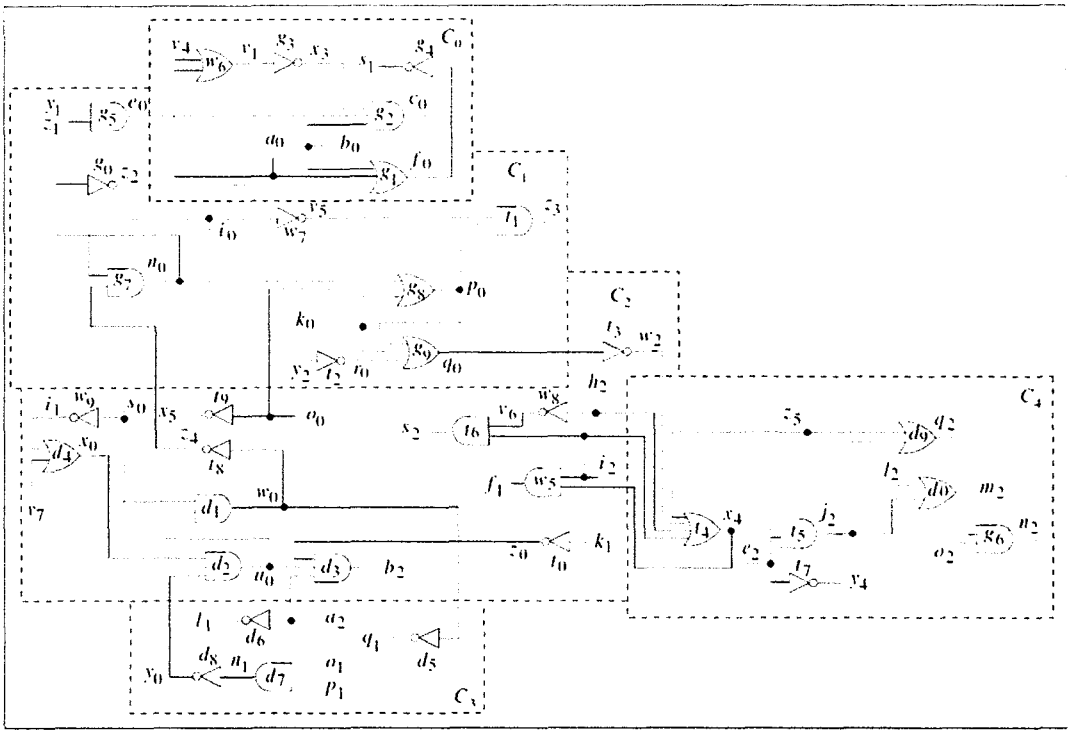


Figure 6.2: The integrated view of the digital system.

Before we show the results of each object individually, let us take a look at the system communications between the objects to have a general view about how the cooperative moralization is done.

Coordinator.CoMoralize() calls agent "G4" to execute CollectMLink().
G4.CollectMLink() calls its compiler to execute DoMoralization().
G4.CollectMLink() calls agent "G2" to execute CollectMLink().
G2.CollectMLink() calls its compiler to execute DoMoralization().
G2.CollectMLink() calls agent "G1" to execute CollectMLink().
G1.CollectMLink() calls its compiler to execute DoMoralization().
G1.CollectMLink() calls agent "G0" to execute CollectMLink().
G0.CollectMLink() calls its compiler to execute DoMoralization().
G0.CollectMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G1".
G1.CollectMLink() receives 12 moral arc(s) from agent "G0":

a0-b0	a0-g1	a0-z2	b0-e0	b0-g1
b0-g2	b0-x3	b0-z2	e0-g2	e0-x3
g1-z2	g2-x3			

G1.CollectMLink() calls its compiler to execute AddToMoralLinks().
G1.CollectMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G2".
G2.CollectMLink() receives 15 moral arc(s) from agent "G1":

g7-i0	g7-z4	g8-k0	g8-n0	g8-o0
g9-k0	g9-p0	g9-r0	i0-z4	k0-n0
k0-o0	k0-r0	n0-o0	p0-r0	t2-y2

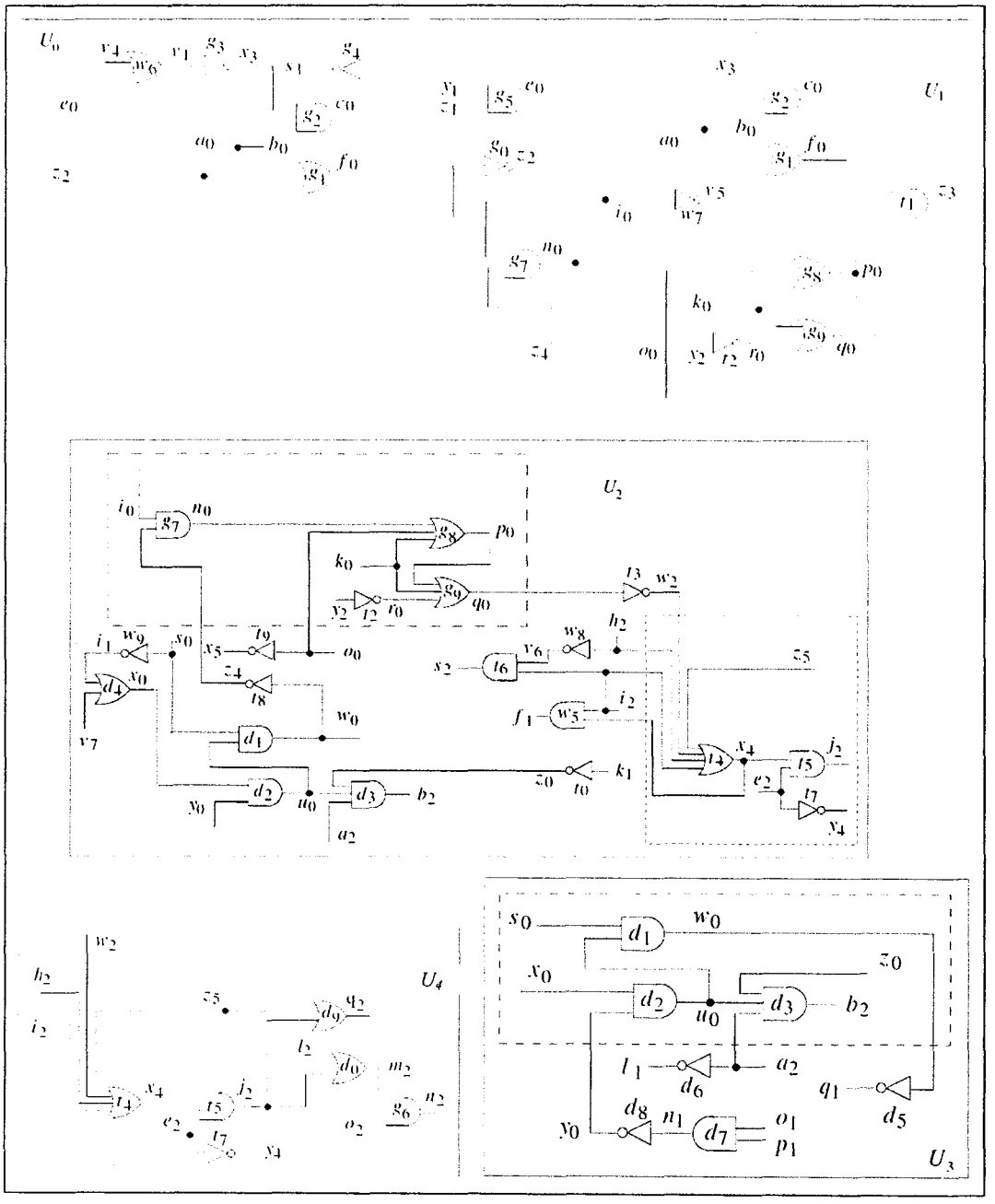


Figure 6.3: The five virtual components of the digital system.

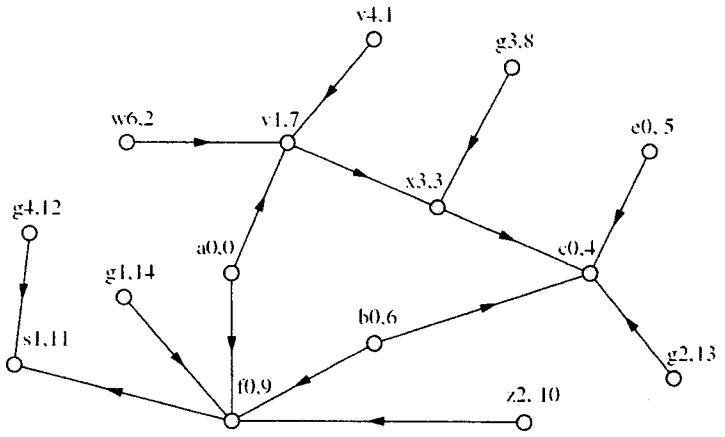


Figure 6.4: The subnet G_0 for virtual component U_0.

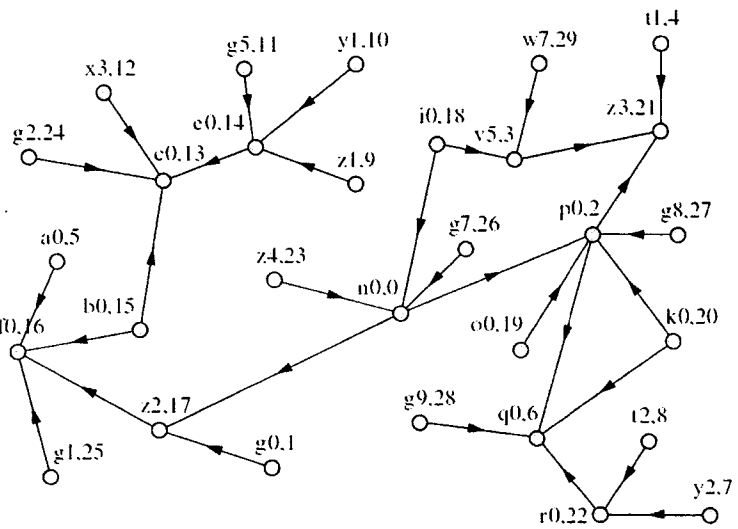


Figure 6.5: The subnet G_1 for virtual component U_1.

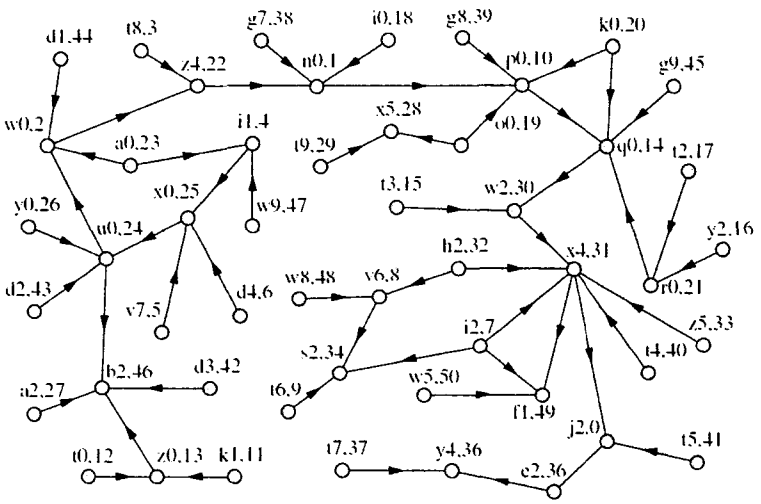


Figure 6.6: The subnet G_2 for virtual component U_2.

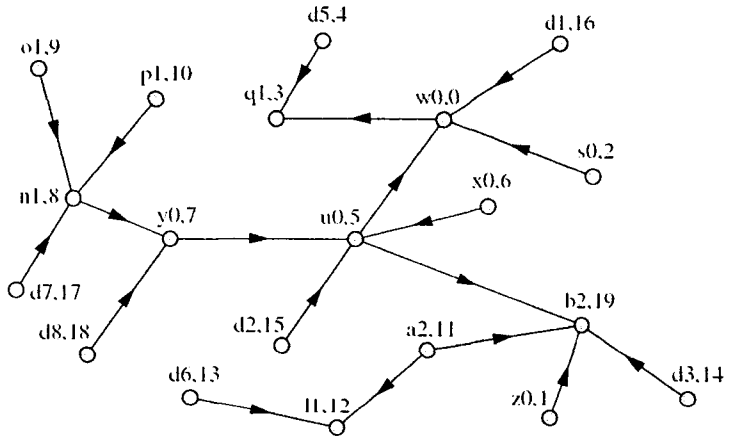


Figure 6.7: The subnet G_3 for virtual component U_3.

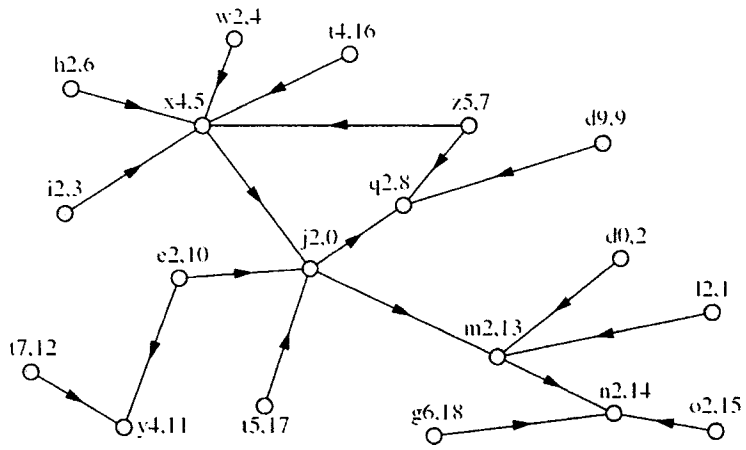


Figure 6.8: The subnet G_4 for virtual component U_4.

G2.CollectMLink() calls its compiler to execute AddToMoralLinks().
G2.CollectMLink() calls agent "G3" to execute CollectMLink().
G3.CollectMLink() calls its compiler to execute DoMoralization().
G3.CollectMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G2".
G2.CollectMLink() receives 12 moral arc(s) from agent "G3":

a2-d3	a2-u0	a2-z0	d1-s0	d1-u0
d2-x0	d2-y0	d3-u0	d3-z0	s0-u0
u0-z0	x0-y0			

G2.CollectMLink() calls its compiler to execute AddToMoralLinks().
G2.CollectMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G4".
G4.CollectMLink() receives 14 moral arc(s) from agent "G2":

e2-t5	e2-t7	e2-x4	h2-i2	h2-t4
h2-w2	h2-z5	i2-t4	i2-w2	i2-z5
t4-w2	t4-z5	t5-x4	w2-z5	

G4.CollectMLink() calls its compiler to execute AddToMoralLinks().
Coordinator.CoMoralize() calls agent "G4" to execute DistributeMLink().
G4.DistributeMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G2".
G4.DistributeMLink() calls agent "G2" to execute DistributeMLink().
G2.DistributeMLink() receives 15 moral arc(s) from agent "G4":

e2-t5	e2-t7	e2-x4	h2-i2	h2-t4
h2-w2	h2-z5	i2-t4	i2-w2	i2-z5
j2-z5	t4-w2	t4-z5	t5-x4	w2-z5

G2.DistributeMLink() calls its compiler to execute AddToMoralLinks().
G2.DistributeMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G1".
G2.DistributeMLink() calls agent "G1" to execute DistributeMLink().
G1.DistributeMLink() receives 15 moral arc(s) from agent "G2":

g7-i0	g7-z4	g8-k0	g8-n0	g8-o0
g9-k0	g9-p0	g9-r0	i0-z4	k0-n0
k0-o0	k0-r0	n0-o0	p0-r0	t2-y2

G1.DistributeMLink() calls its compiler to execute AddToMoralLinks().
G1.DistributeMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G0".
G1.DistributeMLink() calls agent "G0" to execute DistributeMLink().
G0.DistributeMLink() receives 12 moral arc(s) from agent "G1":

a0-b0	a0-g1	a0-z2	b0-e0	b0-g1
b0-g2	b0-x3	b0-z2	e0-g2	e0-x3
g1-z2	g2-x3			

G0.DistributeMLink() calls its compiler to execute AddToMoralLinks().
G2.DistributeMLink() runs GetMoralsWith() to obtain the moral arcs restricted with the intersection with "G3".
G2.DistributeMLink() calls agent "G3" to execute DistributeMLink().
G3.DistributeMLink() receives 12 moral arc(s) from agent "G2":

a2-d3	a2-u0	a2-z0	d1-s0	d1-u0
d2-x0	d2-y0	d3-u0	d3-z0	s0-u0
u0-z0	x0-y0			

G3.DistributeMLink() calls its compiler to execute AddToMoralLinks().

Now we can show the results of the local moralization in each object:

1. For the agent A_0:

Default Arcs:
========

g2-c0	e0-c0	z2-f0	g3-x3	b0-c0
b0-f0	v4-v1	a0-v1	a0-f0	w6-v1
g1-f0	g4-s1	v1-x3	x3-c0	f0-s1

Moral Arcs:
=======

v4-a0	v4-w6	a0-w6	v1-g3	x3-e0
x3-b0	x3-g2	e0-b0	e0-g2	b0-g2
b0-a0	b0-z2	b0-g1	a0-z2	a0-g1
z2-g1	f0-g4			

2. For the agent A_1:

Default Arcs:
========

t1-z3	g8-p0	k0-p0	k0-q0	t2-r0
y2-r0	w7-v5	o0-p0	g9-q0	i0-v5
i0-n0	g7-n0	y1-e0	z1-e0	z4-n0
g0-z2	g5-e0	b0-c0	b0-f0	x3-c0
g2-c0	a0-f0	g1-f0	r0-q0	v5-z3
n0-p0	n0-z2	p0-z3	p0-q0	e0-c0
z2-f0				

Moral Arcs:
=======

y2-t2	i0-w7	i0-z4	i0-g7	z4-g7
n0-o0	n0-k0	n0-g8	o0-k0	o0-g8
k0-g8	v5-p0	v5-t1	p0-t1	p0-r0
p0-g9	k0-r0	k0-g9	r0-g9	y1-z1
y1-g5	z1-g5	x3-e0	x3-b0	x3-g2
e0-b0	e0-g2	b0-g2	n0-g0	b0-a0
b0-z2	b0-g1	a0-z2	a0-g1	z2-g1

3. For the agent A_2:

Default Arcs:
========

k0-p0	k0-q0	g9-q0	t2-r0	y2-r0
z5-x4	t4-x4	t5-j2	e2-y4	e2-j2
t7-y4	w5-f1	i2-s2	i2-x4	i2-f1
h2-v6	h2-x4	t3-w2	o0-p0	o0-x5
g8-p0	i0-n0	g7-n0	t9-x5	w8-v6
t6-s2	k1-z0	t0-z0	d3-b2	a2-b2
d4-x0	v7-x0	d2-u0	w9-i1	y0-u0
s0-i1	s0-w0	t8-z4	d1-w0	z0-b2
i1-x0	x0-u0	u0-b2	u0-w0	w0-z4
z4-n0	n0-p0	p0-q0	v6-s2	r0-q0
q0-w2	w2-x4	x4-f1	x4-j2	

Moral Arcs:
=======

k1-t0	s0-w9	i1-v7	i1-d4	v7-d4
x0-y0	x0-d2	y0-d2	z0-u0	z0-a2
z0-d3	u0-a2	u0-d3	a2-d3	s0-u0
s0-d1	u0-d1	w0-t8	i0-z4	i0-g7
z4-g7	n0-o0	n0-k0	n0-g8	o0-k0
o0-g8	k0-g8	o0-t9	h2-w8	v6-i2
v6-t6	i2-t6	e2-t7	y2-t2	p0-r0
p0-g9	k0-r0	k0-g9	r0-g9	q0-t3
z5-w2	z5-h2	z5-i2	z5-t4	w2-h2
w2-i2	w2-t4	h2-i2	h2-t4	i2-t4
i2-w5	x4-w5	x4-e2	x4-t5	e2-t5
j2-z5				

4. For the agent A_3:

Default Arcs:
========

d3-b2	s0-w0	d1-w0	z0-b2	x0-u0
a2-l1	a2-b2	d5-q1	d2-u0	d6-l1
p1-n1	d8-y0	o1-n1	d7-n1	n1-y0
y0-u0	u0-b2	u0-w0	w0-q1	

Moral Arcs:
=======

a2-d6	o1-p1	o1-d7	p1-d7	n1-d8
x0-y0	x0-d2	y0-d2	z0-u0	z0-a2
z0-d3	u0-a2	u0-d3	a2-d3	s0-u0
s0-d1	u0-d1	w0-d5		

5. For the agent A_4:

Default Arcs:
========

l2-m2	o2-n2	d0-m2	d9-q2	g6-n2
z5-x4	z5-q2	t4-x4	t5-j2	w2-x4
e2-y4	e2-j2	h2-x4	i2-x4	t7-y4
x4-j2	j2-m2	j2-q2	m2-n2	

Moral Arcs:
=======

z5-w2	z5-h2	z5-i2	z5-t4	w2-h2
w2-i2	w2-t4	h2-i2	h2-t4	i2-t4
e2-t7	x4-e2	x4-t5	e2-t5	l2-j2
l2-d0	j2-d0	m2-o2	m2-g6	o2-g6
z5-j2	z5-d9	j2-d9		

6.2.2 Results of the cooperative global triangulation

Here, according to **CoTriangulate** algorithm (Section 3.4.4 on page 37), the system coordinator calls an object arbitrarily to run **DepthFirstEliminate**. After it finishes the coordinator calls the same object to run **DistributeCLinks**.

Before we show the results of each object individually, let us take a look at the system communications between the objects to have a general view about how the cooperative triangulation is done.

Coordinator.CoTriangulate() calls "G4" to run DepthFirstEliminate().
G4.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G2".
G4.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G2": {l2, o2, d0, d9, g6, n2, m2, q2, h2, i2, t4, t5, t7, w2, y4, e2, j2, x4, z5}.
G4.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G2".
G4.DepthFirstEliminate() calls "G2" to run DepthFirstEliminate().
G2.DepthFirstEliminate() receives 0 chordal arc(s) from "G4".
G2.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G2.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G1".
G2.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G1": {t4, t5, t7, w5, t3, t9, w8, t6, k1, t0, d3, a2, d4, v7, d2, w9, y0, t8, d1, z0, b2, x5, s2, v6, h2, y4, e2, f1, i2, j2, z5, x4, w2, i1, x0, s0, u0, w0, g7, g8, g9, i0, o0, q0, t2, y2, r0, k0, p0, n0, z4}.
G2.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G1".
G2.DepthFirstEliminate() calls "G1" to run DepthFirstEliminate().
G1.DepthFirstEliminate() receives 0 chordal arc(s) from "G2".
G1.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G1.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G0".
G1.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G0": {t1, g8, t2, y2, w7, o0, g9, g7, y1, z1, z4, g0, g5, r0, z3, q0, k0, i0, v5, p0, n0, a0, c0, e0, f0, g1, g2, x3, b0, z2}.
G1.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G0".
G1.DepthFirstEliminate() calls "G0" to run DepthFirstEliminate().
G0.DepthFirstEliminate() receives 0 chordal arc(s) from "G1".
G0.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G0.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G1".
G0.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G1": {g3, v4, w6, g4, s1, v1, c0, e0, f0, g1, g2, x3, a0, b0, z2}.
G0.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G1".
G1.DepthFirstEliminate() receives 1 chordal arc(s) from "G0":
a0-x3
G1.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G1.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G2".
G1.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G2": {t1, w7, y1, z1, g0, g5, g2, g1, z3, e0, c0, x3, b0, a0, f0, z2, v5, g7, g8, g9, o0, q0, t2, y2, r0, k0, p0, i0, n0, z4}.
G1.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G2".
G2.DepthFirstEliminate() receives 1 chordal arc(s) from "G1":

i0-p0
G2.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G2.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G3".
G2.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G3": {g9, t2, y2, t4, t5, t7, w5, t3, g8, g7, t9, w8, t6, k1, t0, d4, v7, w9, t8, i1, x5, o0, s2, v6, h2, y4, e2, r0, f1, i2, j2, z5, x4, w2, q0, k0, p0, i0, n0, z4, a2, b2, d1, d2, d3, w0, s0, x0, y0, u0, z0}.
G2.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G3".
G2.DepthFirstEliminate() calls "G3" to run DepthFirstEliminate().
G3.DepthFirstEliminate() receives 1 chordal arc(s) from "G2":
s0-x0
G3.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G3.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G2".
G3.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G2": {d5, d6, p1, d8, o1, d7, l1, n1, q1, a2, b2, d1, d2, d3, w0, s0, x0, y0, u0, z0}.
G3.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G2".
G2.DepthFirstEliminate() receives 1 chordal arc(s) from "G3":
s0-x0
G2.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
G2.DepthFirstEliminate() calls its compiler to run Triangulate() with respect to "G4".
G2.DepthFirstEliminate() saves the resultant elimination sequence in its linkage tree with "G4": {g9, t2, y2, w5, t3, g8, g7, t9, w8, t6, k1, t0, d3, a2, d4, v7, d2, w9, y0, t8, d1, z0, i1, x0, s0, b2, u0, w0, z4, i0, n0, x5, o0, k0, p0, s2, v6, r0, q0, f1, h2, i2, t4, t5, t7, w2, y4, e2, j2, x4, z5}.
G2.DepthFirstEliminate() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G4".
G4.DepthFirstEliminate() receives 0 chordal arc(s) from "G2".
G4.DepthFirstEliminate() calls its compiler to run AddToChordalLinks().
Coordinator.CoTriangulate() calls "G4" to run DistributeCLinks().
G4.DistributeCLinks() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G2".
G4.DistributeCLinks() calls "G2" to run DistributeCLinks().
G2.DistributeCLinks() receives 0 chordal arc(s) from "G4".
G2.DistributeCLinks() calls its compiler to run AddToChordalLinks().
G2.DistributeCLinks() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G1".
G2.DistributeCLinks() calls "G1" to run DistributeCLinks().
G1.DistributeCLinks() receives 1 chordal arc(s) from "G2":
i0-p0
G1.DistributeCLinks() calls its compiler to run AddToChordalLinks().
G1.DistributeCLinks() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G0".
G1.DistributeCLinks() calls "G0" to run DistributeCLinks().
G0.DistributeCLinks() receives 1 chordal arc(s) from "G1":
a0-x3
G0.DistributeCLinks() calls its compiler to run AddToChordalLinks().
G2.DistributeCLinks() runs GetChordalsWith() to obtain the chordal arc(s) restricted with the intersection with "G3".
G2.DistributeCLinks() calls "G3" to run DistributeCLinks().
G3.DistributeCLinks() receives 1 chordal arc(s) from "G2":
s0-x0
G3.DistributeCLinks() calls its compiler to run AddToChordalLinks().

Now we can show the results of the local triangulation in each object:

1. For the agent A_0:

   Elimination Sequence(s):
   ==========

   - with respect to "G1": {g3, v4, w6, g4, v1, s1, c0, e0, f0, g1, g2, x3, a0, b0, z2}.

Chordal Arcs:
========

a0-x3

2. For the agent A_1:

Elimination Sequence(s):
==========

- with respect to "G0": {t1, g8, t2, y2, w7, o0, g9, g7, y1, z1, z4, i0, g0, g5, r0, z3, v5, q0, k0, p0, n0, c0, e0, f0, g1, g2, x3, a0, b0, z2}.
- with respect to "G2": {t1, w7, y1, z1, g0, g5, g2, g1, z3, v5, e0, c0, x3, b0, a0, f0, z2, g7, g8, g9, o0, q0, t2, y2, r0, k0, p0, i0, n0, z4}.

Chordal Arcs:
========

v5-n0 a0-x3 i0-p0

3. For the agent A_2:

Elimination Sequence(s):
==========

- with respect to "G1": {t4, t5, t7, w5, t3, t9, w8, t6, k1, t0, d3, a2, d4, v7, d2, w9, y0, t8, d1, z0, i1, x0, s0, b2, u0, w0, x5, s2, v6, h2, y4, e2, f1, i2, j2, z5, x4, w2, g7, g8, g9, o0, q0, t2, y2, r0, k0, p0, i0, n0, z4}.
- with respect to "G3": {g9, t2, y2, t4, t5, t7, w5, t3, g8, g7, t9, w8, t6, k1, t0, d4, v7, w9, t8, i1, x5, o0, s2, v6, h2, y4, e2, r0, f1, i2, j2, z5, x4, w2, q0, k0, p0, i0, n0, z4, a2, b2, d1, d2, d3, w0, s0, x0, y0, u0, z0}.
- with respect to "G4": {g9, t2, y2, w5, t3, g8, g7, t9, w8, t6, k1, t0, d3, a2, d4, v7, d2, w9, y0, t8, d1, z0, i1, x0, s0, b2, u0, w0, z4, i0, n0, x5, o0, k0, p0, s2, v6, r0, q0, f1, h2, i2, t4, t5, t7, w2, y4, e2, j2, x4, z5}.

Chordal Arcs:
========

s0-x0 i0-p0

4. For the agent A_3:

Elimination Sequence(s):
==========

- with respect to "G2": {d5, d6, p1, d8, o1, d7, l1, n1, q1, a2, b2, d1, d2, d3, w0, s0, x0, y0, u0, z0}.

Chordal Arcs:
========

s0-x0

5. For the agent A_4:

Elimination Sequence(s):
==========

- with respect to "G2": {l2, o2, d0, d9, g6, n2, m2, q2, h2, i2, t4, t5, t7, w2, y4, e2, j2, x4, z5}.

Chordal Arcs:
========

6.2.3 Construction Results of each Object

Immediately after the cooperative triangulation is finished, each object starts constructing its own JT as a hypernode. The following lines record this action:

1. For the agent A_0:

   Generated Cliques and their Initial Potentials:
   ==========

   (a) $[g3, x3, v1] : [P(g3) * P(x3|v1, g3)]$
   (b) $[v1, a0, x3] : [P(a0)]$
   (c) $[v4, v1, a0, w6] : [P(v4) * P(v1|v4, a0, w6) * P(w6)]$
   (d) $[x3, b0, a0] : [P(b0)]$
   (e) $[c0, g2, e0, b0, x3] : [P(c0|x3, e0, b0, g2) * P(g2) * P(e0)]$
   (f) $[f0, z2, b0, a0, g1] : [P(f0|b0, a0, z2, g1) * P(z2) * P(g1)]$
   (g) $[g4, s1, f0] : [P(g4) * P(s1|f0, g4)]$

   DSep Nodes:
   =======

   (a) $[g3, x3, v1] - (x3, v1) - [v1, a0, x3]$
   (b) $[v1, a0, x3] - (v1, a0) - [v4, v1, a0, w6]$
   (c) $[v1, a0, x3] - (a0, x3) - [x3, b0, a0]$
   (d) $[x3, b0, a0] - (x3, b0) - [c0, g2, e0, b0, x3]$
   (e) $[x3, b0, a0] - (b0, a0) - [f0, z2, b0, a0, g1]$
   (f) $[f0, z2, b0, a0, g1] - (f0) - [g4, s1, f0]$

   Linkages with respect to "G1":
   ==========

   (a) $[x3, b0, a0]$ from the host: $[x3, b0, a0]$
   (b) $[c0, g2, e0, b0, x3]$ from the host: $[c0, g2, e0, b0, x3]$
   (c) $[f0, z2, b0, a0, g1]$ from the host: $[f0, z2, b0, a0, g1]$

2. For the agent A_1:

   Generated Cliques and their Initial Potentials:
   ==========

   (a) $[t1, z3, v5, p0] : [P(t1) * P(z3|v5, p0, t1)]$
   (b) $[i0, v5, n0, p0] : [P(i0)]$
   (c) $[w7, v5, i0] : [P(w7) * P(v5|i0, w7)]$
   (d) $[g7, n0, i0, z4] : [P(g7) * P(n0|i0, z4, g7) * P(z4)]$
   (e) $[g8, p0, n0, o0, k0] : [P(g8) * P(p0|n0, o0, k0, g8) * P(o0) * P(k0)]$
   (f) $[g9, q0, p0, k0, r0] : [P(g9) * P(q0|p0, k0, r0, g9)]$
   (g) $[g0, z2, n0] : [P(g0) * P(z2|n0, g0)]$
   (h) $[t2, r0, y2] : [P(t2) * P(r0|y2, t2) * P(y2)]$
   (i) $[f0, b0, a0, g1, z2] : [P(f0|b0, a0, z2, g1) * P(b0) * P(a0) * P(g1)]$
   (j) $[x3, b0, a0] : [P(x3)]$
   (k) $[c0, b0, x3, g2, e0] : [P(c0|x3, e0, b0, g2) * P(g2)]$
   (l) $[y1, e0, z1, g5] : [P(y1) * P(e0|y1, z1, g5) * P(z1) * P(g5)]$

DSep Nodes:
=======

(a) $[t1, z3, v5, p0]$ - $(v5, p0)$ - $[i0, v5, n0, p0]$
(b) $[i0, v5, n0, p0]$ - $(i0, v5)$ - $[w7, v5, i0]$
(c) $[i0, v5, n0, p0]$ - $(i0, n0)$ - $[g7, n0, i0, z4]$
(d) $[i0, v5, n0, p0]$ - $(n0, p0)$ - $[g8, p0, n0, o0, k0]$
(e) $[g8, p0, n0, o0, k0]$ - $(p0, k0)$ - $[g9, q0, p0, k0, r0]$
(f) $[i0, v5, n0, p0]$ - $(n0)$ - $[g0, z2, n0]$
(g) $[g9, q0, p0, k0, r0]$ - $(r0)$ - $[t2, r0, y2]$
(h) $[g0, z2, n0]$ - $(z2)$ - $[f0, b0, a0, g1, z2]$
(i) $[f0, b0, a0, g1, z2]$ - $(b0, a0)$ - $[x3, b0, a0]$
(j) $[x3, b0, a0]$ - $(x3, b0)$ - $[c0, b0, x3, g2, e0]$
(k) $[c0, b0, x3, g2, e0]$ - $(e0)$ - $[y1, e0, z1, g5]$

Linkages with respect to "G0":
==========

(a) $[f0, b0, a0, g1, z2]$ from the host: $[f0, b0, a0, g1, z2]$
(b) $[x3, b0, a0]$ from the host: $[x3, b0, a0]$
(c) $[c0, b0, x3, g2, e0]$ from the host: $[c0, b0, x3, g2, e0]$

Linkages with respect to "G2":
==========

(a) $[i0, n0, p0]$ from the host: $[i0, v5, n0, p0]$
(b) $[g7, n0, i0, z4]$ from the host: $[g7, n0, i0, z4]$
(c) $[g8, p0, n0, o0, k0]$ from the host: $[g8, p0, n0, o0, k0]$
(d) $[g9, q0, p0, k0, r0]$ from the host: $[g9, q0, p0, k0, r0]$
(e) $[t2, r0, y2]$ from the host: $[t2, r0, y2]$

3. For the agent A_2:

Generated Cliques and their Initial Potentials:
==========

(a) $[t4, x4, z5, w2, h2, i2] : [P(t4) * P(x4|z5, w2, h2, i2, t4) * P(z5) * P(h2) * P(i2)]$
(b) $[w5, f1, i2, x4] : [P(w5) * P(f1|i2, x4, w5)]$
(c) $[v6, h2, i2] : []$
(d) $[j2, x4, z5] : []$
(e) $[w8, v6, h2] : [P(w8) * P(v6|h2, w8)]$
(f) $[t6, s2, v6, i2] : [P(t6) * P(s2|v6, i2, t6)]$
(g) $[t5, j2, x4, e2] : [P(t5) * P(j2|x4, e2, t5) * P(e2)]$
(h) $[t3, w2, q0] : [P(t3) * P(w2|q0, t3)]$
(i) $[t7, y4, e2] : [P(t7) * P(y4|e2, t7)]$
(j) $[g9, q0, p0, k0, r0] : [P(g9) * P(q0|p0, k0, r0, g9) * P(k0)]$
(k) $[g8, p0, n0, o0, k0] : [P(g8) * P(p0|n0, o0, k0, g8) * P(o0)]$
(l) $[p0, n0, i0] : [P(i0)]$
(m) $[g7, n0, i0, z4] : [P(g7) * P(n0|i0, z4, g7)]$
(n) $[t2, r0, y2] : [P(t2) * P(r0|y2, t2) * P(y2)]$
(o) $[t9, x5, o0] : [P(t9) * P(x5|o0, t9)]$

(p) $[t8, z4, w0] : [P(t8) * P(z4|w0, t8)]$

(q) $[d1, w0, s0, u0] : [P(d1) * P(w0|s0, u0, d1) * P(s0)]$

(r) $[x0, u0, s0] : []$

(s) $[i1, s0, x0] : []$

(t) $[d2, u0, x0, y0] : [P(d2) * P(u0|x0, y0, d2) * P(y0)]$

(u) $[d4, x0, i1, v7] : [P(d4) * P(x0|i1, v7, d4) * P(v7)]$

(v) $[w9, i1, s0] : [P(w9) * P(i1|s0, w9)]$

(w) $[d3, b2, z0, u0, a2] : [P(d3) * P(b2|z0, u0, a2, d3) * P(a2)]$

(x) $[k1, z0, t0] : [P(k1) * P(z0|k1, t0) * P(t0)]$

DSep Nodes:
=======

(a) $[t4, x4, z5, w2, h2, i2]$ - $(x4, i2)$ - $[w5, f1, i2, x4]$

(b) $[t4, x4, z5, w2, h2, i2]$ - $(h2, i2)$ - $[v6, h2, i2]$

(c) $[t4, x4, z5, w2, h2, i2]$ - $(x4, z5)$ - $[j2, x4, z5]$

(d) $[v6, h2, i2]$ - $(v6, h2)$ - $[w8, v6, h2]$

(e) $[v6, h2, i2]$ - $(v6, i2)$ - $[t6, s2, v6, i2]$

(f) $[j2, x4, z5]$ - $(j2, x4)$ - $[t5, j2, x4, e2]$

(g) $[t4, x4, z5, w2, h2, i2]$ - $(w2)$ - $[t3, w2, q0]$

(h) $[t5, j2, x4, e2]$ - $(e2)$ - $[t7, y4, e2]$

(i) $[t3, w2, q0]$ - $(q0)$ - $[g9, q0, p0, k0, r0]$

(j) $[g9, q0, p0, k0, r0]$ - $(p0, k0)$ - $[g8, p0, n0, o0, k0]$

(k) $[g8, p0, n0, o0, k0]$ - $(p0, n0)$ - $[p0, n0, i0]$

(l) $[p0, n0, i0]$ - $(n0, i0)$ - $[g7, n0, i0, z4]$

(m) $[g9, q0, p0, k0, r0]$ - $(r0)$ - $[t2, r0, y2]$

(n) $[g8, p0, n0, o0, k0]$ - $(o0)$ - $[t9, x5, o0]$

(o) $[g7, n0, i0, z4]$ - $(z4)$ - $[t8, z4, w0]$

(p) $[t8, z4, w0]$ - $(w0)$ - $[d1, w0, s0, u0]$

(q) $[d1, w0, s0, u0]$ - $(s0, u0)$ - $[x0, u0, s0]$

(r) $[x0, u0, s0]$ - $(x0, s0)$ - $[i1, s0, x0]$

(s) $[x0, u0, s0]$ - $(x0, u0)$ - $[d2, u0, x0, y0]$

(t) $[i1, s0, x0]$ - $(i1, x0)$ - $[d4, x0, i1, v7]$

(u) $[i1, s0, x0]$ - $(i1, s0)$ - $[w9, i1, s0]$

(v) $[d1, w0, s0, u0]$ - $(u0)$ - $[d3, b2, z0, u0, a2]$

(w) $[d3, b2, z0, u0, a2]$ - $(z0)$ - $[k1, z0, t0]$

Linkages with respect to "G1":
==========

(a) $[g9, q0, p0, k0, r0]$ from the host: $[g9, q0, p0, k0, r0]$

(b) $[g8, p0, n0, o0, k0]$ from the host: $[g8, p0, n0, o0, k0]$

(c) $[p0, n0, i0]$ from the host: $[p0, n0, i0]$

(d) $[g7, n0, i0, z4]$ from the host: $[g7, n0, i0, z4]$

(e) $[t2, r0, y2]$ from the host: $[t2, r0, y2]$

Linkages with respect to "G3":
==========

(a) $[d1, w0, s0, u0]$ from the host: $[d1, w0, s0, u0]$

(b) $[x0, u0, s0]$ from the host: $[x0, u0, s0]$

(c) $[d2, u0, x0, y0]$ from the host: $[d2, u0, x0, y0]$

(d) $[d3, b2, z0, u0, a2]$ from the host: $[d3, b2, z0, u0, a2]$

Linkages with respect to "G4":
==========

(a) $[t4, x4, z5, w2, h2, i2]$ from the host: $[t4, x4, z5, w2, h2, i2]$

(b) $[j2, x4, z5]$ from the host: $[j2, x4, z5]$

(c) $[t5, j2, x4, e2]$ from the host: $[t5, j2, x4, e2]$

(d) $[t7, y4, e2]$ from the host: $[t7, y4, e2]$

4. For the agent A_3:

Generated Cliques and their Initial Potentials:
==========

(a) $[d5, q1, w0] : [P(d5) * P(q1|w0, d5)]$

(b) $[d1, w0, s0, u0] : [P(d1) * P(w0|s0, u0, d1) * P(s0)]$

(c) $[s0, u0, x0] : [P(x0)]$

(d) $[d2, u0, x0, y0] : [P(d2) * P(u0|x0, y0, d2)]$

(e) $[a2, b2, z0, u0, d3] : [P(a2) * P(b2|z0, u0, a2, d3) * P(z0) * P(d3)]$

(f) $[d8, y0, n1] : [P(d8) * P(y0|n1, d8)]$

(g) $[d6, l1, a2] : [P(d6) * P(l1|a2, d6)]$

(h) $[p1, n1, o1, d7] : [P(p1) * P(n1|o1, p1, d7) * P(o1) * P(d7)]$

DSep Nodes:
==========

(a) $[d5, q1, w0]$ - $(w0)$ - $[d1, w0, s0, u0]$

(b) $[d1, w0, s0, u0]$ - $(s0, u0)$ - $[s0, u0, x0]$

(c) $[s0, u0, x0]$ - $(u0, x0)$ - $[d2, u0, x0, y0]$

(d) $[d1, w0, s0, u0]$ - $(u0)$ - $[a2, b2, z0, u0, d3]$

(e) $[d2, u0, x0, y0]$ - $(y0)$ - $[d8, y0, n1]$

(f) $[a2, b2, z0, u0, d3]$ - $(a2)$ - $[d6, l1, a2]$

(g) $[d8, y0, n1]$ - $(n1)$ - $[p1, n1, o1, d7]$

Linkages with respect to "G2":
==========

(a) $[d1, w0, s0, u0]$ from the host: $[d1, w0, s0, u0]$

(b) $[s0, u0, x0]$ from the host: $[s0, u0, x0]$

(c) $[d2, u0, x0, y0]$ from the host: $[d2, u0, x0, y0]$

(d) $[a2, b2, z0, u0, d3]$ from the host: $[a2, b2, z0, u0, d3]$

5. For the agent A_4:

Generated Cliques and their Initial Potentials:
==========

(a) $[l2, m2, j2, d0] : [P(l2) * P(m2|l2, j2, d0) * P(d0)]$

(b) $[o2, n2, m2, g6] : [P(o2) * P(n2|m2, o2, g6) * P(g6)]$

(c) $[d9, q2, z5, j2] : [P(d9) * P(q2|z5, j2, d9) * P(z5)]$

(d) $[j2, x4, z5] : []$

(e) $[t5, j2, x4, e2] : [P(t5) * P(j2|x4, e2, t5) * P(e2)]$

(f) $[h2, x4, z5, w2, i2, t4] : [P(h2) * P(x4|z5, w2, h2, i2, t4) * P(w2) * P(i2) * P(t4)]$

(g) $[t7, y4, e2] : [P(t7) * P(y4|e2, t7)]$

DSep Nodes:
=======

(a) $[l2, m2, j2, d0]$ - $(m2)$ - $[o2, n2, m2, g6]$

(b) $[l2, m2, j2, d0]$ - $(j2)$ - $[d9, q2, z5, j2]$

(c) $[d9, q2, z5, j2]$ - $(z5, j2)$ - $[j2, x4, z5]$

(d) $[j2, x4, z5]$ - $(j2, x4)$ - $[t5, j2, x4, e2]$

(e) $[j2, x4, z5]$ - $(x4, z5)$ - $[h2, x4, z5, w2, i2, t4]$

(f) $[t5, j2, x4, e2]$ - $(e2)$ - $[t7, y4, e2]$

Linkages with respect to "G2":
==========

(a) $[j2, x4, z5]$ from the host: $[j2, x4, z5]$

(b) $[t5, j2, x4, e2]$ from the host: $[t5, j2, x4, e2]$

(c) $[h2, x4, z5, w2, i2, t4]$ from the host: $[h2, x4, z5, w2, i2, t4]$

(d) $[t7, y4, e2]$ from the host: $[t7, y4, e2]$

6.2.4 Belief Updating to Bring the LJF into Consistence

By the end of the preceding step, the MSDAG of the digital system has been transformed into its corresponding LJF model. The last step before applying any observation is to bring the multi-agent reasoning system belief into consistence. That can be accomplished by running **CommunicateBelief** algorithm (Section 4.3 on page 45).

First we present the conversations between agents and then we show clearly the consistence between agents' belief by normalizing the variables' belief in each agent. The belief equality of each variable $V \in \{I_i\}$ in all agents ($V \in T_i$) reflects the global consistency.

Coordinator.CommunicateBelief() calls "G4" to run CollectBelief().
G4.CollectBelief() calls "G2" to run CollectBelief().
G2.CollectBelief() calls "G1" to run CollectBelief().
G1.CollectBelief() calls "G0" to run CollectBelief().
G0.CollectBelief() runs UnifyBelief().
G1.CollectBelief() runs UpdateBelief() with respect to "G0".
  G1.UpdateBelief() calls "G0" to run AssignLinkagePotentials().
    G0.AssignLinkagePotentials() calls its linkage tree with "G1" to run PreparePotentials().
  G1.UpdateBelief() calls its linkage tree with "G0" to run Absorb().
  G1.UpdateBelief() runs UnifyBelief().
G2.CollectBelief() runs UpdateBelief() with respect to "G1".
  G2.UpdateBelief() calls "G1" to run AssignLinkagePotentials().
    G1.AssignLinkagePotentials() calls its linkage tree with "G2" to run PreparePotentials().
  G2.UpdateBelief() calls its linkage tree with "G1" to run Absorb().
  G2.UpdateBelief() runs UnifyBelief().

G2.CollectBelief() calls "G3" to run CollectBelief().
G3.CollectBelief() runs UnifyBelief().
G2.CollectBelief() runs UpdateBelief() with respect to "G3".
  G2.UpdateBelief() calls "G3" to run AssignLinkagePotentials().
    G3.AssignLinkagePotentials() calls its linkage tree with "G2" to run PreparePotentials().
  G2.UpdateBelief() calls its linkage tree with "G3" to run Absorb().
  G2.UpdateBelief() runs UnifyBelief().
G4.CollectBelief() runs UpdateBelief() with respect to "G2".
  G4.UpdateBelief() calls "G2" to run AssignLinkagePotentials().
    G2.AssignLinkagePotentials() calls its linkage tree with "G4" to run PreparePotentials().
  G4.UpdateBelief() calls its linkage tree with "G2" to run Absorb().
  G4.UpdateBelief() runs UnifyBelief().
Coordinator.CommunicateBelief() calls "G4" to run DistributeBelief().
G4.DistributeBelief() calls "G2" to run DistributeBelief().
G2.DistributeBelief() runs UpdateBelief() with respect to "G4".
  G2.UpdateBelief() calls "G4" to run AssignLinkagePotentials().
    G4.AssignLinkagePotentials() calls its linkage tree with "G2" to run PreparePotentials().
  G2.UpdateBelief() calls its linkage tree with "G4" to run Absorb().
  G2.UpdateBelief() runs UnifyBelief().
G2.DistributeBelief() calls "G1" to run DistributeBelief().
G1.DistributeBelief() runs UpdateBelief() with respect to "G2".
  G1.UpdateBelief() calls "G2" to run AssignLinkagePotentials().
    G2.AssignLinkagePotentials() calls its linkage tree with "G1" to run PreparePotentials().
  G1.UpdateBelief() calls its linkage tree with "G2" to run Absorb().
  G1.UpdateBelief() runs UnifyBelief().
G1.DistributeBelief() calls "G0" to run DistributeBelief().
G0.DistributeBelief() runs UpdateBelief() with respect to "G1".
  G0.UpdateBelief() calls "G1" to run AssignLinkagePotentials().
    G1.AssignLinkagePotentials() calls its linkage tree with "G0" to run PreparePotentials().
  G0.UpdateBelief() calls its linkage tree with "G1" to run Absorb().
  G0.UpdateBelief() runs UnifyBelief().
G2.DistributeBelief() calls "G3" to run DistributeBelief().
G3.DistributeBelief() runs UpdateBelief() with respect to "G2".
  G3.UpdateBelief() calls "G2" to run AssignLinkagePotentials().
    G2.AssignLinkagePotentials() calls its linkage tree with "G3" to run PreparePotentials().
  G3.UpdateBelief() calls its linkage tree with "G2" to run Absorb().
  G3.UpdateBelief() runs UnifyBelief().

- For the agent A_0:

$P(g2 =' good'\|\{\}) = 0.990000$	$P(e0 =' zero'\|\{\}) = 0.992000$	$P(z2 =' zero'\|\{\}) = 0.974494$
$P(g3 =' good'\|\{\}) = 0.990000$	$P(b0 =' zero'\|\{\}) = 1.000000$	$P(v4 =' zero'\|\{\}) = 0.000000$
$P(a0 =' zero'\|\{\}) = 1.000000$	$P(w6 =' good'\|\{\}) = 0.990000$	$P(g1 =' good'\|\{\}) = 0.990000$
$P(g4 =' good'\|\{\}) = 0.990000$	$P(v1 =' zero'\|\{\}) = 0.003000$	$P(x3 =' zero'\|\{\}) = 0.992030$
$P(c0 =' zero'\|\{\}) = 0.992000$	$P(f0 =' zero'\|\{\}) = 0.971647$	$P(s1 =' zero'\|\{\}) = 0.033069$

- For the agent A_1:

$P(t1 =' good'\|\{\}) = 0.990000$	$P(g8 =' good'\|\{\}) = 0.990000$	$P(k0 =' zero'\|\{\}) = 1.000000$
$P(t2 =' good'\|\{\}) = 0.990000$	$P(y2 =' zero'\|\{\}) = 0.000000$	$P(w7 =' good'\|\{\}) = 0.990000$
$P(o0 =' zero'\|\{\}) = 1.000000$	$P(g9 =' good'\|\{\}) = 0.990000$	$P(i0 =' zero'\|\{\}) = 0.000000$
$P(g7 =' good'\|\{\}) = 0.990000$	$P(y1 =' zero'\|\{\}) = 0.000000$	$P(z1 =' zero'\|\{\}) = 1.000000$
$P(z4 =' zero'\|\{\}) = 0.012920$	$P(g0 =' good'\|\{\}) = 0.990000$	$P(g5 =' good'\|\{\}) = 0.990000$
$P(b0 =' zero'\|\{\}) = 1.000000$	$P(x3 =' zero'\|\{\}) = 0.992030$	$P(g2 =' good'\|\{\}) = 0.990000$
$P(a0 =' zero'\|\{\}) = 1.000000$	$P(g1 =' good'\|\{\}) = 0.990000$	$P(r0 =' zero'\|\{\}) = 0.995000$
$P(v5 =' zero'\|\{\}) = 0.995000$	$P(n0 =' zero'\|\{\}) = 0.020713$	$P(p0 =' zero'\|\{\}) = 0.023589$
$P(z3 =' zero'\|\{\}) = 0.987196$	$P(q0 =' zero'\|\{\}) = 0.026330$	$P(e0 =' zero'\|\{\}) = 0.992000$
$P(c0 =' zero'\|\{\}) = 0.992000$	$P(z2 =' zero'\|\{\}) = 0.974494$	$P(f0 =' zero'\|\{\}) = 0.971647$

- For the agent A_2:

$P(k0 =' zero'|\{\}) = 1.000000$
$P(y2 =' zero'|\{\}) = 0.000000$
$P(t5 =' good'|\{\}) = 0.990000$
$P(w5 =' good'|\{\}) = 0.990000$
$P(t3 =' good'|\{\}) = 0.990000$
$P(i0 =' zero'|\{\}) = 0.000000$
$P(w8 =' good'|\{\}) = 0.990000$
$P(t0 =' good'|\{\}) = 0.990000$
$P(d4 =' good'|\{\}) = 0.990000$
$P(w9 =' good'|\{\}) = 0.990000$
$P(t8 =' good'|\{\}) = 0.990000$
$P(i1 =' zero'|\{\}) = 0.005000$
$P(b2 =' zero'|\{\}) = 0.036053$
$P(n0 =' zero'|\{\}) = 0.020713$
$P(v6 =' zero'|\{\}) = 0.005000$
$P(r0 =' zero'|\{\}) = 0.995000$
$P(x4 =' zero'|\{\}) = 0.003000$
$P(g9 =' good'|\{\}) = 0.990000$
$P(z5 =' zero'|\{\}) = 1.000000$
$P(e2 =' zero'|\{\}) = 0.000000$
$P(i2 =' zero'|\{\}) = 0.000000$
$P(o0 =' zero'|\{\}) = 1.000000$
$P(g7 =' good'|\{\}) = 0.990000$
$P(t6 =' good'|\{\}) = 0.990000$
$P(d3 =' good'|\{\}) = 0.990000$
$P(v7 =' zero'|\{\}) = 0.000000$
$P(y0 =' zero'|\{\}) = 0.012920$
$P(d1 =' good'|\{\}) = 0.990000$
$P(x0 =' zero'|\{\}) = 0.003000$
$P(w0 =' zero'|\{\}) = 0.992000$
$P(p0 =' zero'|\{\}) = 0.023589$
$P(s2 =' zero'|\{\}) = 0.012920$
$P(q0 =' zero'|\{\}) = 0.026330$
$P(f1 =' zero'|\{\}) = 0.010952$
$P(t2 =' good'|\{\}) = 0.990000$
$P(t4 =' good'|\{\}) = 0.990000$
$P(t7 =' good'|\{\}) = 0.990000$
$P(h2 =' zero'|\{\}) = 1.000000$
$P(g8 =' good'|\{\}) = 0.990000$
$P(t9 =' good'|\{\}) = 0.990000$
$P(k1 =' zero'|\{\}) = 1.000000$
$P(a2 =' zero'|\{\}) = 0.000000$
$P(d2 =' good'|\{\}) = 0.990000$
$P(s0 =' zero'|\{\}) = 1.000000$
$P(z0 =' zero'|\{\}) = 0.005000$
$P(u0 =' zero'|\{\}) = 0.023627$
$P(z4 =' zero'|\{\}) = 0.012920$
$P(x5 =' zero'|\{\}) = 0.005000$
$P(y4 =' zero'|\{\}) = 0.995000$
$P(w2 =' zero'|\{\}) = 0.968933$
$P(j2 =' zero'|\{\}) = 0.010952$

- For the agent A_3:

$P(d3 =' good'|\{\}) = 0.990000$
$P(z0 =' zero'|\{\}) = 0.005000$
$P(d5 =' good'|\{\}) = 0.990000$
$P(p1 =' zero'|\{\}) = 0.000000$
$P(d7 =' good'|\{\}) = 0.990000$
$P(y0 =' zero'|\{\}) = 0.012920$
$P(w0 =' zero'|\{\}) = 0.992000$
$P(s0 =' zero'|\{\}) = 1.000000$
$P(x0 =' zero'|\{\}) = 0.003000$
$P(d2 =' good'|\{\}) = 0.990000$
$P(d8 =' good'|\{\}) = 0.990000$
$P(l1 =' zero'|\{\}) = 0.995000$
$P(u0 =' zero'|\{\}) = 0.023627$
$P(q1 =' zero'|\{\}) = 0.012920$
$P(d1 =' good'|\{\}) = 0.990000$
$P(a2 =' zero'|\{\}) = 0.000000$
$P(d6 =' good'|\{\}) = 0.990000$
$P(o1 =' zero'|\{\}) = 1.000000$
$P(n1 =' zero'|\{\}) = 0.992000$
$P(b2 =' zero'|\{\}) = 0.036053$

- For the agent A_4:

$P(l2 =' zero'|\{\}) = 1.000000$
$P(d9 =' good'|\{\}) = 0.990000$
$P(t4 =' good'|\{\}) = 0.990000$
$P(e2 =' zero'|\{\}) = 0.000000$
$P(t7 =' good'|\{\}) = 0.990000$
$P(j2 =' zero'|\{\}) = 0.010952$
$P(q2 =' zero'|\{\}) = 0.013886$
$P(o2 =' zero'|\{\}) = 0.000000$
$P(g6 =' good'|\{\}) = 0.990000$
$P(t5 =' good'|\{\}) = 0.990000$
$P(h2 =' zero'|\{\}) = 1.000000$
$P(x4 =' zero'|\{\}) = 0.003000$
$P(m2 =' zero'|\{\}) = 0.013886$
$P(d0 =' good'|\{\}) = 0.990000$
$P(z5 =' zero'|\{\}) = 1.000000$
$P(w2 =' zero'|\{\}) = 0.968933$
$P(i2 =' zero'|\{\}) = 0.000000$
$P(y4 =' zero'|\{\}) = 0.995000$
$P(n2 =' zero'|\{\}) = 0.021664$

6.2.5 Processing Observations in the System

Suppose that the gates d_1 in digital component C_2 and t_5 in C_4 are faulty and produce incorrect outputs. The state of the total universe is then completely defined and is shown in Figure 6.9. The input and output of each gate are labeled in the figure. Because gates d_1 and t_5 produce incorrect outputs, the outputs of other gates downstream will also be affected. Each incorrect output is underlined in the figure.

These observations can be injected in the multi-agent reasoning system as evidence using the **EnterEvidence** algorithm (Section 2.5.6 on page 26) by agents A_2 and A_4. **CommunicateBelief** needs to be called to propagate e-messages and return the system into consistence again. Here we show the significant changes in each agent that is affected by the two instantiated variables ($d_1 =' bad'$, $t_5 =' bad'$):

Note that these observations are seen as evidence only by their agents where they were entered, they are not seen as evidence in the others.

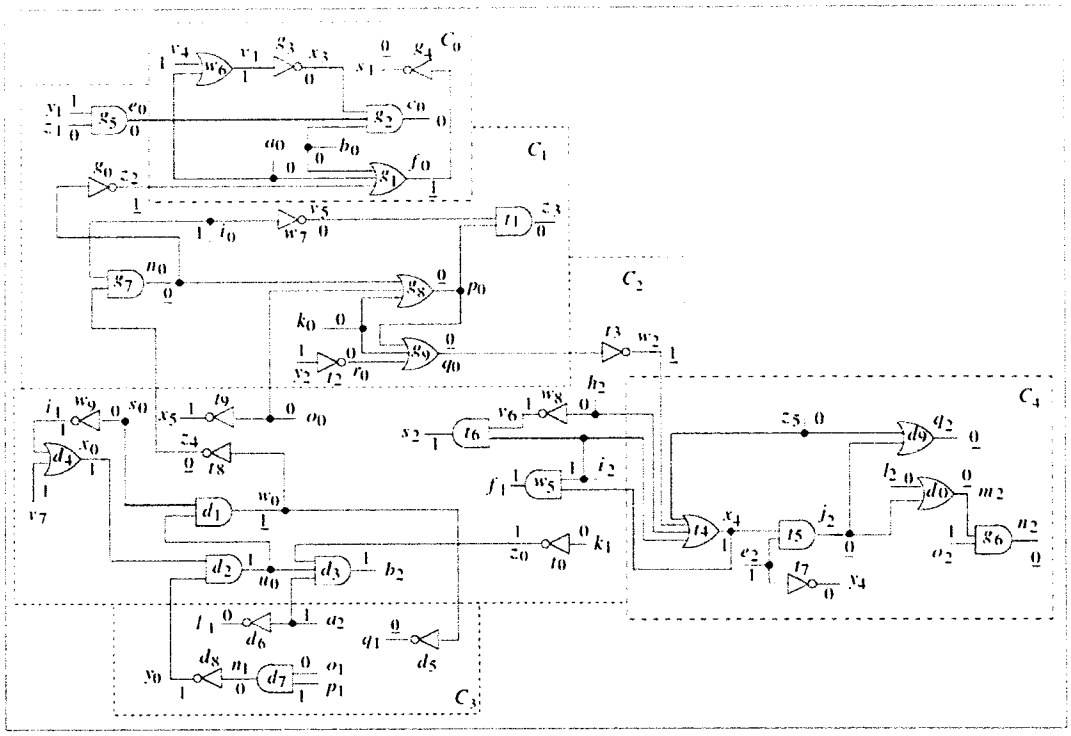


Figure 6.9: The inputs and outputs of all gates with incorrect outputs underlined.

- For the agent A_0:

$P(z2 =' zero'|\{\}) = 0.210675$ $P(f0 =' zero'|\{\}) = 0.212411$ $P(s1 =' zero'|\{\}) = 0.784713$

- For the agent A_1:

$P(z4 =' zero'|\{\}) = 0.796999$ $P(n0 =' zero'|\{\}) = 0.792247$ $P(p0 =' zero'|\{\}) = 0.790494$
$P(q0 =' zero'|\{\}) = 0.784822$ $P(z2 =' zero'|\{\}) = 0.210675$ $P(f0 =' zero'|\{\}) = 0.212411$

- For the agent A_2:

$P(t5 =' good'|\{d1 =' bad'\}) = 0.000000$ $P(d1 =' good'|\{d1 =' bad'\}) = 0.000000$
$P(w0 =' zero'|\{d1 =' bad'\}) = 0.200001$ $P(z4 =' zero'|\{d1 =' bad'\}) = 0.796999$
$P(n0 =' zero'|\{d1 =' bad'\}) = 0.792247$ $P(p0 =' zero'|\{d1 =' bad'\}) = 0.790494$
$P(q0 =' zero'|\{d1 =' bad'\}) = 0.784822$ $P(w2 =' zero'|\{d1 =' bad'\}) = 0.218026$
$P(j2 =' zero'|\{d1 =' bad'\}) = 0.798200$

- For the agent A_3:

$P(d1 =' good'|\{\}) = 0.000000$ $P(w0 =' zero'|\{\}) = 0.200001$ $P(q1 =' zero'|\{\}) = 0.796999$

- For the agent A_4:

$P(t5 =' good'|\{t5 =' bad'\}) = 0.000000$ $P(w2 =' zero'|\{t5 =' bad'\}) = 0.218026$
$P(j2 =' zero'|\{t5 =' bad'\}) = 0.798200$ $P(m2 =' zero'|\{t5 =' bad'\}) = 0.796411$
$P(n2 =' zero'|\{t5 =' bad'\}) = 0.791668$ $P(q2 =' zero'|\{t5 =' bad'\}) = 0.796411$

Conclusions and Future Work

In Chapters 3 and 4, we studied in detail why a set of agents over a large and complex domain should be organized into an MSBN and how. We studied how they can perform probabilistic reasoning exactly, effectively, and distributively. Then we provide an effective cooperative method to assign shared variables' belief in an MSBN in order to satisfy Xiang's 5^{th} assumption in the case of distributed multi-agent reasoning systems.

In Chapter 5, first we gave an overview about feed-forward artificial neural networks and how they can be trained using a back-propagation training algorithm, and then we showed how ANN can play a role in a multi-agent reasoning system to enhance the communication speed between agents. In this thesis, much programming has been done (more than 3000 lines). We spent a full term just to prepare a very consistent package that has no limitation but the available memory space. In spite of everything, it was a very good experience to prepare such an advanced system from scratch especially since the package has many practical applications.

The results that have been shown in Chapters 4 through 6 are very complete, transparent and verifiable, and the time taken is acceptable. Having a very good and deep knowledge about such a topic motivates us to go forward in this research and try to provide more contributions. We have started to build a real distributed multi-agent reasoning system by using the internet as a communication media but unfortunately we don't have enough time.

In the future, if a chance of obtaining a PhD admission, some ideas will be studied and then implemented. One idea is studying and then employing the TCP/IP network protocol to establish a direct client/server reliable connection between each pair of hypernodes to carry e-messages between them. Another idea involves giving the system coordinator just enough more privileges to allow the building of a dynamic multi-agent reasoning system that can be changed dynamically, all without compromising each agent's privacy.

References

[1] Bayes's theorem. http://en.wikipedia.org/wiki/Bayes%27_theorem, January 2007.

[2] R. Brunelli and T. Poggio. Face recognition: Features versus templates. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(10):1042–1052, 1993.

[3] T. Caetano. *Graphical Models and Point Set Matching.* PhD thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, July 2004.

[4] G. Csaba. Creation of a bayesian network-based meta spam filter, using the analysis of different spam filters. Master's thesis, University of Copenhagen, May 2006.

[5] M. Druzdzel and M. Hernion. Intercausal reasoning with uninstantiated ancestor nodes. In *Ninth annual conference on uncertainty in artifecial intelligence*, UAI-93, pages 317–325, Washington, D.C., July 1993.

[6] R. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction.* Addison Wesley Publishing Company, Reading, MA, 1989.

[7] D. Heckerman. A tutorial on learning bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, March 1995.

[8] Yu Hen Hu and Jenq-Neng Hwang, editors. *HandBook of Neural Network Signal Processing.* CRC Press, September 2002.

[9] A. Paz J. Tian and J. Pearl. Finding minimal d-separators. Technical Report R-254, University of California, February 1998.

[10] F. Jensen. *Bayesian Networks and Decision Graphs.* Springer, 2002.

[11] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quaterly*, 4:269–282, 1990.

[12] Finn Jensen and Frank Jensen. Optimal junction trees. In *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 360–366, San Francisco, CA, 1994. Morgan Kaufmann.

[13] V. Kecman. *Learning and Soft Computing: Support Vector Machines, Neural Networks, and Fuzzy Logic Models.* MIT Press, Cambridge, MA, 2001.

[14] U. Kjaerulff and A. Madsen. Probabilistic networks: An introduction to bayesian networks and influence diagrams. May 2005.

[15] G. F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Pearson Education Limited, Essex, England, forth edition, 2002.

[16] R. Neapolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms.* John Wiley & Sons, New York, NY, 1990.

[17] R. Neapolitan. *Learning Bayesian Networks.* Prentice Hall, New York, NY, 2003.

[18] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, San Mateo, California, 1988.

[19] Valluru B. Rao. *C++ Neural Networks and Fuzzy Logic.* M&T Books, IDG Books Worldwide, Inc., June 1995.

[20] H. Simon. *Why Should Machines Learn?, Machine Learning: An Artificial Intelligence Approach*, volume I. Tioga Pub. Co., Palo Alto, CA, 1983.

[21] Y. Xiang. *Probabilistic Reasoning in Multiagent Systems: A Graphical Models Approach.* Cambridge University Press, New York, NY, 2002.

[22] Y. Xiang and V. Lesser. Justifying multiply sectioned bayesian networks. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 349–356, Boston, MA, July 2000.

[23] Y. Xiang, D. Poole, and M. P. Beddoes. Multiply sectioned bayesian networks and junction forests for large knowledge-based systems. *Computational Intelligence*, 9:171–220, 1993.