Inference in Multiply Sectioned Bayesian Networks: Methods and Performance Comparison

Y. Xiang
University of Guelph, Canada, yxiang@cis.uoguelph.ca
F.V. Jensen
Aalborg University, Denmark, fvj@cs.auc.dk
X. Chen
University of Guelph, Canada, xiaoyun@cis.uoguelph.ca

***Abstract*— We extend lazy propagation for inference in single-agent Bayesian networks to multiagent lazy inference in multiply sectioned Bayesian networks (MSBNs). Two methods are proposed using distinct runtime structures. We prove that the new methods are exact and efficient when domain structure is sparse. Both improve space and time complexity than the existing method, which allow multiagent probabilistic reasoning to be performed in much larger domains given the computational resource. Relative performance of the three methods are compared analytically and experimentally.**

I. Introduction

Multiply Sectioned Bayesian Networks (MSBNs) [20] extend Bayesian networks (BNs) [8] for modular and flexible knowledge representation and inference. Although they were originally motivated within the single-agent paradigm [19], their modularity allows natural extension into the multiagent paradigm [13]. From a small set of meta-requirements, (1) exact probability measure of agent belief, (2) agent communication by belief over small sets of shared variables, (3) a simpler organization of agents, (4) a directed acyclic graph (DAG) domain dependence structure, and (5) the joint belief of agents admitting individual belief on internal variables and combining their beliefs on shared variables, it has been shown [18] that the resultant representation of a cooperative multiagent system is an MSBN. These meta-requirements distinguish MSBNs from a number of alternative knowledge representations which do not simultaneously satisfy these meta-requirements [18]. The multiagent paradigm will be followed in this paper.

The first general inference method in MSBNs [20], [11] was an extension of a junction tree (JT) based inference method [5] for single-agent BNs. We shall refer to this method as the *product-based inference with linked junction forest (LJF)* and we overview the method later in the paper. The method allows exact and autonomous inference in a cooperative multiagent system that is efficient when the dependence structure is sparse. An agent's inference is *autonomous* if it can be performed by the agent independently without communication with other agents, and after the inference the agent is able to answer probabilistic queries exactly conditioned on all local knowledge and observations and on all global knowledge and observations up to the last communication. See [14] for a comparison of methods regarding autonomy.

The product-based inference with LJF has been compared [14] with extensions of other inference methods for single-agent BNs, in particular, the loop-cutset methods and two stochastic sampling methods. The comparison shows that product-based inference with LJF is superior than these alternatives. In this paper, we present two new general inference methods for MSBNs which extend the lazy propagation [6] for single-agent BNs. The two methods will be referred to as *lazy inference with double-linked junction forest (DLJF)* and *lazy inference with LJF*, and we will explain later the corresponding terminologies. The two new methods are also exact and autonomous, and they promise to deliver improved run-time inference efficiency, which we will demonstrate with experimental results.

We assume that readers are familiar with common terminologies used in the literature of Bayesian networks, such as DAGs, mapping of domain variables and nodes in a BN, parent, child and family of a node in a BN, d-separation, moralization, node elimination and fill-ins, triangulation and cliques, junction trees, clusters and separators in a JT, conditional independence, conditional probability tables, a potential (non-normalized probability distribution) and its domain, consistence of potentials, message passing in JTs, etc. Definitions of these terminologies can be found in a number of reference books such as

[8], [7], [1], [2], [4], [13].

The remainder of the paper is organized as follows: Basics of MSBNs are introduced in Section II. Section III overviews product-based inference with LJF and Section IV overviews lazy propagation. Section V presents lazy inference with DLJF and it is an extension of [17]. Section VI presents lazy inference with LJF and it is an extension of [16]. Experimental comparison of the three methods is reported in Section VII. Section VIII draws conclusions from this work.

II. Overview of Multiply Sectioned Bayesian Networks

An MSBN M is a collection of Bayesian subnets, one from each agent, that together defines a BN. M represents probabilistic dependence of a *domain* partitioned into multiple *subdomains* each of which is represented by a subnet. Agents cooperate to reason about the state of the domain in order to take proper actions. Without confusion, we refer to an agent, its subdomain, and its subnet interchangeably. To ensure exact and autonomous inference, subnets are required to satisfy certain conditions [18] described below:

Given a graph $G = (N, E)$, a partition of N into N_0 and N_1 such that $N_0 \cup N_1 = N$ and $N_0 \cap N_1 \neq \emptyset$, and subgraphs G_i of G spanned by N_i $(i = 0, 1)$, G is said to be *sectioned* into G_0 and G_1. A multi-subdomain graphical model is defined based on sectioning:

Definition 1: Let $G = (N, E)$ be a connected graph sectioned into subgraphs $\{G_i = (N_i, E_i)\}$. Let the subgraphs be organized into an undirected tree Ψ where each node is uniquely labeled by a G_i and each link between G_k and G_m is labeled by the non-empty `interface` $N_k \cap N_m$ such that for each i and j, $N_i \cap N_j$ is contained in each subgraph on the path between G_i and G_j in Ψ. Then Ψ is a `hypertree` over G. Each G_i is a `hypernode` and each interface is a `hyperlink`. A pair of hypernodes connected by a hyperlink is said to be `adjacent`.

Each hyperlink serves as the information channel between subnets connected and is referred to as an agent *interface*. Agents communicate by exchanging beliefs over their interfaces. An interface must be a *d-sepset*, as defined below:

Definition 2: Let G be a directed graph such that a hypertree over G exists. A node x (whose parent set in G, possibly empty, is denoted $\pi(x)$) contained in more than one subgraph in G is a `d-sepnode` if there exists at least one subgraph that contains $\pi(x)$. An interface I is a `d-sepset` if every $x \in I$ is a d-sepnode.

The overall structure of an MSBN is a hypertree MSDAG:

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

Graphically, a hyperlink separates the hypertree MSDAG into two subtrees. Semantically, this corresponds to conditional independence given the d-sepset. An MSBN is then defined as follows:

Definition 4: An MSBN M is a triplet $M = (\mathcal{N}, G, \mathcal{P})$. $\mathcal{N} = \bigcup_i N_i$ is the `domain` where each N_i is a set of variables. $G = \bigsqcup_i G_i$ (a hypertree MSDAG) is the `structure` where nodes of each DAG G_i are labeled by elements of N_i. Let x be a variable and $\pi(x)$ be all the parents of x in G. For each x, exactly one of its occurrences (in a G_i containing $\{x\} \cup \pi(x)$) is assigned $P(x|\pi(x))$, and each occurrence in other DAGs is assigned a constant table. $\mathcal{P} = \prod_i P_i(N_i)$ is the `jpd` (joint probability distribution), where each $P_i(N_i)$ is the product of probability tables associated with nodes in G_i. Each triplet $S_i = (N_i, G_i, P_i)$ 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 on the hypertree MSDAG.

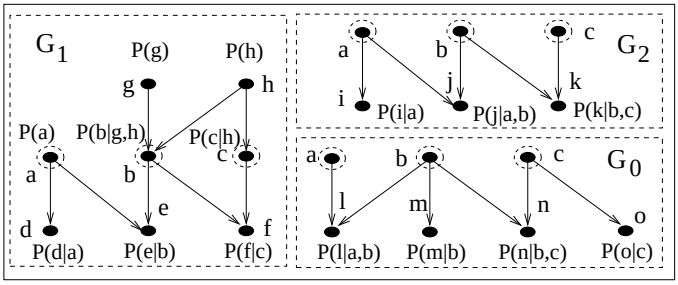

Fig. 1. A trivial MSBN with three subnets and hypertree structure $G_1 - G_0 - G_2$. The two d-sepsets are identical and each consists of $\{a, b, c\}$. Each d-sepnode in each subgraph is shown with a dashed circle.

An example MSBN is shown in Figure 1.

III. Product-Based Inference with Linked Junction Forest

Product-based inference with LJF conducts probabilistic reasoning in an MSBN by message passing. Each message is a potential over a subset of variables. *Local inference* within each agent passes intra-subnet messages which bring a subnet into consistency. *Communication* among agents passes inter-subnet messages which brings the multiagent system into global consistency.

To pass these messages efficiently for exact inference, each agent compiles its subnet into a JT. The compilation

is a cooperative process [12]. First, agents perform distributed moralization and triangulation. Each agent then constructs its local JT. This is followed by local potential assignment: For each variable in the agent's subnet, its conditional probability table is associated with a cluster in the JT that contains the variable and its parents in the subnet. Each cluster in each JT is assigned with a single potential over its set of member variables, that is the product of all conditional probability tables associated with the cluster. Hence, we name this inference method *product-based*.

For communication with adjacent agents, an agent compiles its d-sepset with an adjacent agent into a *linkage tree*. The linkage tree is derived from the agent's local JT as detailed in [11]. We outline the key properties of linkage tree here as they are essential to the results of this paper. Note that an agent has a single subnet and compiles it into a single JT. If the agent is adjacent to k agents on the hypertree, then it is associated with k d-sepsets and will compile them into k linkage trees.

1) A linkage tree is a JT over a d-sepset, where each cluster is a subset of the d-sepset, called a *linkage*, and each separator is called a *linkage separator*.
2) Given a JT of an agent and one of its linkage trees, each cluster in the linkage tree is a subset of at least one cluster in the JT. One such cluster is designated as the *linkage host* of the linkage.
3) A linkage tree expresses the same graphical separation within the d-sepset as its deriving JT. Hence, a linkage tree encodes identical independence relations within the d-sepset as the corresponding JT. Furthermore, a d-sepset involves two adjacent agents, the linkage tree derived by one agent is equivalent to that derived by the other.

Figure 2 illustrates JTs and linkage trees obtained from the MSBN in Figure 1.

Once the linkage trees are compiled, the structure of the MSBN has been compiled into a set of local JTs related by linkage trees. For each pair of adjacent agents, their local JTs are linked by a linkage tree. Such a runtime structure is called a *linked junction forest*.

The last step of the compilation is belief initialization. System wide communication is performed for agents to exchange prior knowledge on shared variables. Communication consists of one round of inward message propagation along the hypertree and one round of outward propagation. The message sent from an agent to an adjacent agent consists of a set of linkage potentials as detailed in [11]. After belief initialization, each agent is able to perform autonomous inference and to answer probabilistic queries exactly relative to prior knowledge of all agents and observations of its own.

IV. Overview of Lazy Propagation

Lazy propagation [6] is an inference method for single-agent BNs based on message passing in a JT of the BN. Each cluster is assigned a set of probability tables from the BN. Unlike product-based inference in [5], the product of these tables are not obtained. We refer to these tables as potentials, refer to the cluster of current focus by C, and refer to the set of potentials at C by β. When no potential is assigned to a cluster, $\beta = \emptyset$. The *joint system potential* of the JT over a domain N is defined as the product of potentials in all clusters, denoted by $B(N)$. In the following, we describe data structures and algorithms for lazy propagation.

Each separator S between two adjacent clusters C and C' is associated with two buffers. One buffer stores message from C' to C and the other from C to C'. For the given cluster C and separator S, we refer to the two buffers as *in-buffer* and *out-buffer*, respectively, relative to C. A cluster executes the following algorithm to compute and send a message to an adjacent cluster, where $\setminus$ is the set difference operator.

Algorithm 1—SendPotential: Let C be a cluster with β. Let adjacent clusters be $C_1, ..., C_m$. Let β_i be the set of potentials in the in-buffer from C_i. When SendPotential relative to C_k is called in C, C does the following:

(1) $\beta' = \beta \cup_{i \neq k} \beta_i$.

(2) Marginalize out variables $C \setminus C_k$ from β'. (To marginalize out variable x, multiply potentials with x in the domain and apply marginalization to the product.)

(3) Send the resultant set of potentials to the out-buffer to C_k.

In the following two algorithms, C is a cluster and `caller` is an adjacent cluster or the JT. Algorithm 2 is executed recursively by each cluster during inward message passing.

Algorithm 2—CollectPotential: When CollectPotential is called in cluster C, C does the following:

(1) For each adjacent cluster Q except caller, call CollectPotential in Q and receive potentials from Q.

(2) SendPotential relative to caller if it is an adjacent cluster.

Algorithm 3 is executed recursively by each cluster during outward message passing.

Algorithm 3—DistributePotential: When DistributePotential is called in C, for each adjacent cluster Q except caller, C performs the following:

(1) SendPotential relative to Q.

(2) Call DistributePotential in Q.

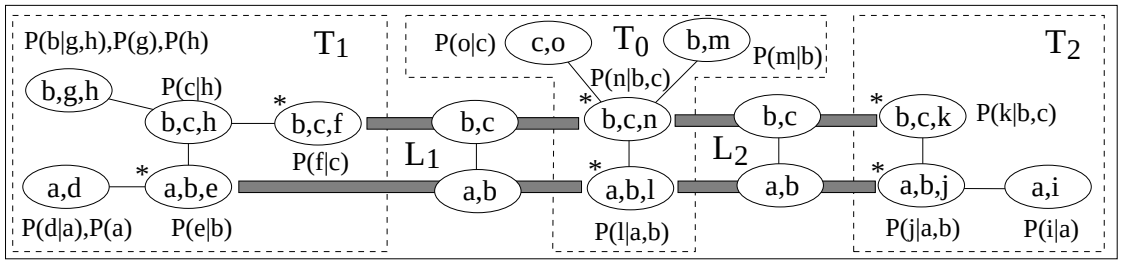

Fig. 2. JTs and linkage trees obtained from Figure 1. T_i $(i = 0, 1, 2)$ is the JT obtained from subnet G_i. L_1 is the linkage tree between T_0 and T_1 and L_2 between T_0 and T_2. Next to each cluster are the probability tables assigned to it. Each linkage host is labeled by *. Each thick link relates a linkage to its host in a local JT.

Algorithm 4 is executed by a JT, which produces the exact marginal for each cluster.

Algorithm 4—UnifyPotential:

(1) Select a cluster C arbitrarily.

(2) Call CollectPotential in C.

(3) Call DistributePotential in C.

Proposition 1 summarizes the effect of UnifyPotential, where $const$ denotes a constant:

Proposition 1—Proposition 3.4 in [10]: Let UnifyPotential be performed in a JT. For any cluster C with β and in-buffer message β_i from separator R_i $(i = 1, ..., m)$, denote the product of potentials in β as $\beta(C)$ and the product of potentials in β_i as $\beta_i(R_i)$. Then $\beta(C)\ \prod_{i=1}^{m} \beta_i(R_i) = const \sum_{N \setminus C} B(N)$.

V. Lazy Inference with Double-Linked Junction Forest

The advantage of lazy propagation over product-based inference is the space efficiency gained by replacing product-based potentials and messages with factorized ones. We extend single-agent lazy propagation to multiagent MSBNs for space efficiency and refer to the resultant method as *lazy inference*. Lazy propagation employs only two numerical operations on potentials: multiplication and marginalization, while product-based inference in single-agent JT and in multiagent LJF also requires division. In this section, we present a runtime structure for lazy inference that uses only multiplication and marginalization. It is referred to as *Double-Linked Junction Forest* (DLJF).

A. Cooperative Triangulation

The order in which messages, both intra-subnet and inter-subnet, are produced during inference can be determined by triangulation. For agent privacy and flexibility in operation, a cooperative triangulation (versus centralized) is preferred. Similar to cooperative triangulation used for product-based inference in LJF [12], we present a triangulation consisting of two rounds of fill-in propagation along the hypertree. We first illustrate the process using the example MSBN in Figure 3 (a) with its hypertree in (c) and its moral graph in (b). The agent organization is isomorphic to (c), which is obtained by substituting each G_i with the agent A_i. This example will be used in subsequent sections.

The first round is the inward triangulation. We assume the root agent A_1. Instead of using different notations for local DAG, moral graph and chordal graph, we denote all of them by G and differentiate them by context. Processing starts from leaf agents A_3 and A_0. To triangulate local moral graph G_3, A_3 eliminates nodes outside d-sepset with A_2, namely, nodes n and m. Suppose the order is (n, m), which produces fill-ins $\{\{j, k\}, \{j, l\}\}$ shown in (d) as dashed links. The resultant chordal graph is labeled $G_{3\to2}$ to signify that it is used to compute message from A_3 to A_2 during inference. Completion of d-sepset $\{j, k, l\}$ indicates that during lazy inference, message from A_3 to A_2 may contain a potential over $\{j, k, l\}$. Elimination order (n, m) is one possible order for marginalization in computing that message. To ensure that A_2 have the proper data structure to process this message, A_3 sends the above fill-ins to A_2. Similarly, A_0 eliminates $\{o, p\}$ in order (o, p). This produces fill-in $\{f, j\}$ shown in (e), which A_0 sends to A_2.

After receiving fill-ins from A_3 and A_0, A_2 performs triangulation of G_2 augmented with incoming fill-ins. It eliminates nodes outside d-sepset with A_1, namely, $\{i, j, k, l\}$ in the order (l, k, j, i). This produces fill-ins $\{\{h, j\}, \{f, h\}\}$ shown in (f). A_2 sends fill-in $\{f, h\}$ to A_1 (how A_1 makes use of the fill-in is discussed in Section V-C). Since A_1 is the root, inward triangulation ends.

The second round is outward triangulation starts from A_1. It eliminates nodes outside the d-sepset with A_2 from G_1 in the order (a, b, c, d, e). The resultant $G_{1\to2}$ is shown in (g). No fill-ins are produced and none is sent to A_2. Note that the d-sepset $\{f, g, h\}$ is *not* complete (compare with $G_{2\to1}$). Hence, message from A_1 to A_2 never con-

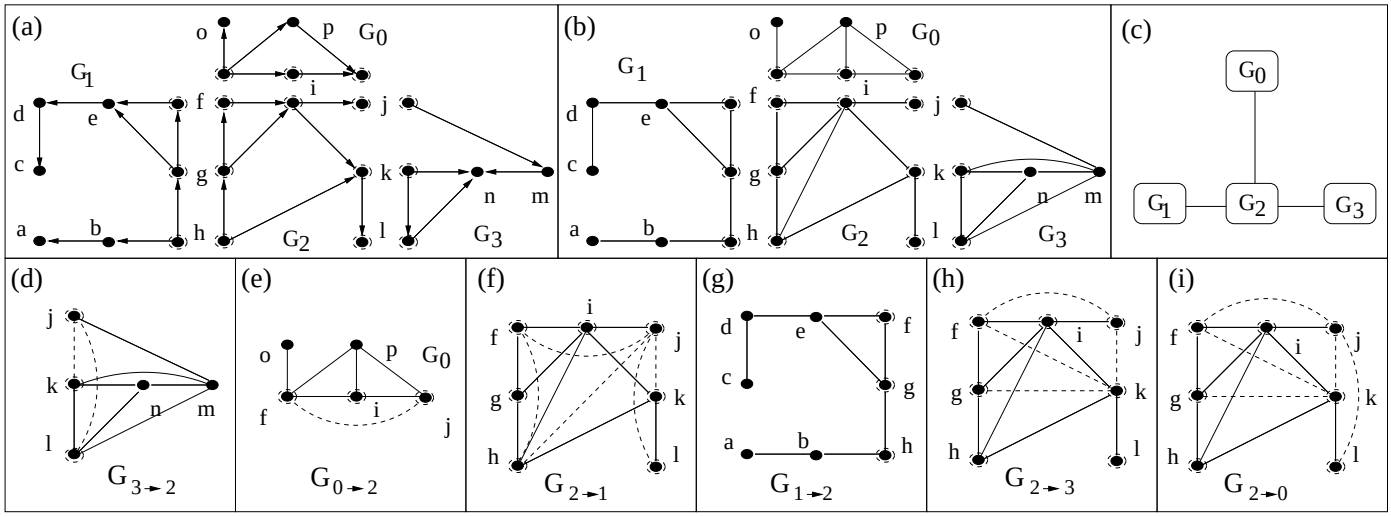

Fig. 3. Cooperative triangulation. (a) An example MSBN whose hypertree is shown in (c). (b) The moral graph of the MSBN in (a). (c) The hypertree structure of the MSBN. (d) The local graph after inward triangulation at G_3. (e) The local graph after inward triangulation at G_0. (f) The local graph after inward triangulation at G_2. (g) The local graph after outward triangulation at G_1. (h) The local graph after outward triangulation at G_2 relative to G_3. (i) The local graph after outward triangulation at G_2 relative to G_0.

tains a potential over the entire d-sepset.

Next, A_2 triangulates G_2 with respect to A_3 and A_0. The two operations can be paralleled. With respect to A_3, after adding the fill-in $\{f, j\}$ from A_0, A_2 eliminates $\{f, g, h, i\}$ in the order (h, g, f, i) producing fill-ins $\{\{g, k\}, \{f, k\}, \{j, k\}\}$ as shown in (h). The fill-in on d-sepset, $\{j, k\}$, is sent to A_3. Note that the d-sepset is again not complete.

With respect to A_0, after adding fill-ins $\{\{j, k\}, \{j, l\}\}$ from A_3, A_2 eliminates $\{g, h, k, l\}$ in the order (l, h, g, k) producing fill-ins $\{\{g, k\}, \{f, k\}, \{f, j\}\}$ as shown in (i). A_2 then sends A_0 the fill-in $\{f, j\}$. Since A_0 and A_3 are leaf subnets in the hypertree, cooperative triangulation ends.

Next, we present general algorithms of which the above illustration is a trace. We say that a graph G over N is *triangulated in order* O, if N is eliminated in O with fill-ins added to G. Given a set F of links over a set N of nodes, a subset $E \subseteq F$ is called a *restriction* of F to $X \subset N$ if

$$E = \{\{x, y\} | x \in X,\ y \in X,\ \{x, y\} \in F\},$$

and we denote $E = F^{\downarrow X}$. Algorithm 5 is executed recursively by each agent during inward triangulation. The agent who executes is referred to as A_0. A_c denotes the `caller` which is either the system coordinator or an adjacent agent of A_0. Other adjacent agents of A_0 are denoted as $A_1, ..., A_m$.

Algorithm 5: [CollectFillin] Let $G_i = (N_i, E_i)$ be the local moral graph of A_i. When A_0 is called by A_c to perform CollectFillin, A_0 does the following:

initialize accumulator $F = \phi$;
for $i = 1$ to m, do in parallel
  call CollectFillin in A_i and receive fill-ins F_i on d-sepset $N_i \cap N_0$;
  update $F = F \cup F_i$;
if A_c is an adjacent agent, do
  eliminate $N_0 \setminus N_c$ from $G'_0 = (N_0, E_0 \cup F)$ in an order $O_{0 \to c}$ and add fill-ins to F;
  denote $G_{0 \to c} = (N_0, E_0 \cup F)$ and send $F^{\downarrow N_0 \cap N_c}$ to A_c;

CollectFillin does not restrict order $O_{0 \to c}$. Proposition 2 shows that fill-ins passed between agents are independent of elimination order.

Proposition 2: Fill-ins $F^{\downarrow N_0 \cap N_c}$ produced by CollectFillin is independent of orders in which eliminations are performed.

Proof: We prove by induction. Suppose A_0 is a leaf on hypertree. The *for* loop is skipped and elimination is performed directly in G_0. Consider a pair of nodes $x, y \in N_0 \cap N_c$ that are not directly connected in G_0. When $N_0 \setminus N_c$ is eliminated, fill-in $\{x, y\}$ will be added iff there is a path between x and y such that all nodes on the path (except x and y) are in $N_0 \setminus N_c$ [9] (Lemma 4). Hence the proposition holds.

If A_0 is not a leaf, assume that the proposition holds for F_i $(i = 1, ..., m)$. Then G'_0 is independent of the order in which eliminations are performed in each subtree rooted at A_i. Using the same argument above to elimination in G'_0, the proposition is proven. □

Algorithm 6 is executed recursively by each agent during outward triangulation.

Algorithm 6: [DistributeFillin] Let $G_i = (N_i, E_i)$ be the local graph produced by CollectFillin at A_i, and F_i be fill-ins that A_0 received from A_i. When A_0 is called by A_c to perform DistributeFillin with fill-ins F_c, A_0 does the following:

for $i = 1$ to m, do
  denote $F = F_c \cup_{k=1\ (k\neq i)}^{m} F_k$;
  eliminate $N_0 \setminus N_i$ from $(N_0, E_0 \cup F)$ in an order $O_{0\rightarrow i}$ and add fill-ins to F;
  denote $G_{0\rightarrow i} = (N_0, E_0 \cup F)$;
  call DistributeFillin in A_i with fill-ins $F^{\downarrow N_0 \cap N_i}$;

Fill-ins passed between agents during DistributeFillin are also independent of the elimination orders as stated in Proposition 3. It can be proven similarly to Proposition 2.

Proposition 3: Fill-ins $F^{\downarrow N_0 \cap N_i}$ produced by DistributeFillin is independent of the orders in which eliminations are performed.

Algorithm 7 combines the above algorithms for cooperative triangulation of an MSBN.

Algorithm 7: [CommunicateFillin]
(1) Select an agent A arbitrarily.
(2) Call CollectFillin in A.
(3) Call DistributeFillin in A with empty fill-ins.

The illustration presented early in this section is the trace of CommunicateFillin with $A = A_1$. Any agent may be selected as the root in CommunicateFillin. Will the choice of root agent affect the outcome? Theorem 1 answers this question.

Theorem 1: Fill-ins $F^{\downarrow N \cap N_i}$ sent by each agent during CommunicateFillin is independent of the root agent being selected.

Proof: During CommunicateFillin, each agent sends fill-ins to each adjacent agent exactly once. Consider an agent Ag with an adjacent agent Ag'. Denote the sub-hypertree rooted at Ag away from Ag' by T and denote the sub-hypertree rooted at Ag' away from Ag by T'. If A is in T, fill-ins sent from Ag to Ag' are produced during DistributeFillin. If A is in T', fill-ins sent from Ag to Ag' are produced during CollectFillin.

In either case, only subnets in T are relevant. From Propositions 2 and 3, the result follows. □

B. Run Time Structure for Message Computation

The chordal graphs produced above provide *implicit* structures for computing messages in lazy inference. Each chordal graph is then organized into a set of JTs, called a junction forest (JF), for message computation. We illustrate by continuing with the example.

Consider $G_{3\rightarrow 2}$ in Figure 3 (d). Since the d-sepset is complete, we organize cliques of $G_{3\rightarrow 2}$ into a JT $T_{3\rightarrow 2}$ shown in Figure 4 (1). That is, each cluster of $T_{3\rightarrow 2}$ is a clique in $G_{3\rightarrow 2}$. During inference, message from A_3 to A_2 will be obtained from cluster $\{j, k, l, m\}$. Similarly, JTs $T_{0\rightarrow 2}$ (in Figure 4 (2)), $T_{2\rightarrow 1}$ (in (3)) and $T_{2\rightarrow 0}$ (in (4)) are created from $G_{0\rightarrow 2}$ (in Figure 3 (e)), $G_{2\rightarrow 1}$ (in (f)) and $G_{2\rightarrow 0}$ (in (i)), respectively.

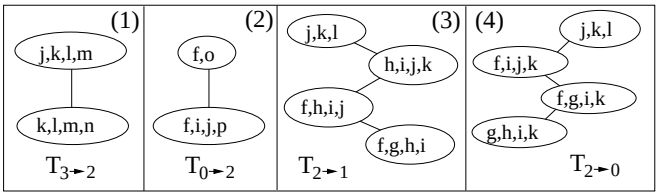

Fig. 4. Message JFs when d-sepset is complete. (1) The JT obtained from $G_{3\rightarrow 2}$ in Figure 3 (d). (2) The JT obtained from $G_{0\rightarrow 2}$ in Figure 3 (e). (3) The JT obtained from $G_{2\rightarrow 1}$ in Figure 3 (f). (4) The JT obtained from $G_{2\rightarrow 0}$ in Figure 3 (i).

Next, consider $G_{1\rightarrow 2}$ in Figure 3 (g). Since the d-sepset is incomplete, message from A_1 to A_2 may contain potentials over $\{f, g\}$ or $\{g, h\}$, but not $\{f, g, h\}$. A JF of two JTs is then created, shown (one at the upper right and the other at the lower left) in Figure 5 (1), and message will be obtained from clusters $\{e, f, g\}$ and $\{g, h\}$.

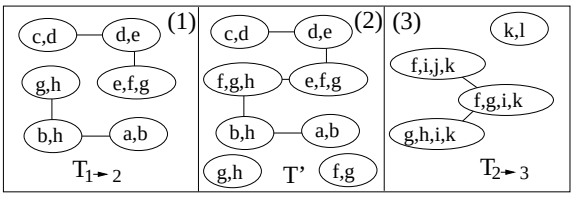

Fig. 5. Message JFs when d-sepset is incomplete. (1) The JF obtained from $G_{1\rightarrow 2}$ in Figure 3 (g). (2) The intermediate JF used to build the JF in (1). (3) The JF obtained from $G_{2\rightarrow 3}$ in Figure 3 (h).

To build this JF, we create the JF shown in Figure 5 (2). For each clique in the subgraph of $G_{1\rightarrow 2}$ spanned by the d-sepset, create a cluster. This produces two isolated clusters at the bottom. Complete the d-sepset in $G_{1\rightarrow 2}$ and create the JT shown on the top of (2). Delete cluster $\{f, g, h\}$ made of the d-sepset, breaking the JT into subtrees. In one subtree, cluster $\{b, h\}$ was adjacent to $\{f, g, h\}$. Since the isolated cluster $\{g, h\}$ satisfies $\{g, h\} \cap \{b, h\} = \{g, h\} \cap \{f, g, h\}$, we connect $\{g, h\}$ with $\{b, h\}$. For the other subtree, cluster $\{e, f, g\}$ was adjacent to $\{f, g, h\}$. Since the isolated cluster $\{f, g\}$ is a subset of $\{e, f, g\}$, we remove $\{f, g\}$. The resultant JF is in (1). Similarly, JF $T_{2\rightarrow 3}$ (Figure 5 (3)) is created from $G_{2\rightarrow 3}$ (Figure 3 (h)).

Below is the general algorithm. A subset of nodes is *eliminable* if they can be eliminated without fill-ins.

Algorithm 8: [BuildMessageJF] Let A_0 be an agent and A_1 an adjacent agent over N_1. Let $G_{0\rightarrow 1}$ be the chordal graph at A_0 over N_0 such that $N_0 \setminus N_1$ is eliminable. When A_0 is called to BuildMessageJF relative to

A_1, A_0 does the following:
identify set $L_{0\to1}$ of cliques in subgraph of $G_{0\to1}$ spanned by $N_0 \cap N_1$;
if $L_{0\to1}$ is a singleton, create a JT $T_{0\to1}$ from cliques of $G_{0\to1}$ and halt;

complete $N_0 \cap N_1$ in $G_{0\to1}$ and denote resultant graph by G';
create a JT T' from G';
create a JF $T_{0\to1}$ consisting of T' and cliques (disconnected) in $L_{0\to1}$;
remove cluster $N_0 \cap N_1$ from T', breaking T' into subtrees;
for each subtree of T' originally rooted at cluster $N_0 \cap N_1$ with adjacent cluster C', do
find a clique C from $L_{0\to1}$ such that $C' \cap C = C' \cap (N_0 \cap N_1)$;
if $C \subset C'$, remove C from $T_{0\to1}$;
else connect C to C';

Proposition 4 shows that a message JF can always be constructed by BuildMessageJF.

Proposition 4: Junction forest $T_{0\to1}$ from BuildMessageJF is well defined.

Proof: $G_{0\to1}$ is obtained from CommunicateFillin. Hence, it is chordal and $N_0 \setminus N_1$ is eliminable. Thus, if $L_{0\to1}$ is a singleton, a JT can be constructed and BuildMessageJF ends at the first half.

If $L_{0\to1}$ is not a singleton, $G_{0\to1}$ with completion of $N_0 \cap N_1$ is chordal since $N_0 \setminus N_1$ is eliminable. Hence, T' exists and $N_0 \cap N_1$ is a cluster in T'. We only have to show that for each C' a C can be found.

C' contains at least one node outside $N_0 \cap N_1$. Completion of $N_0 \cap N_1$ does not affect connectivity of such nodes. Hence, C' is a clique in $G_{0\to1}$ where $C' \cap (N_0 \cap N_1)$ is complete. It follows that a clique $C \subset (N_0 \cap N_1)$ exists in $G_{0\to1}$ such that $C' \cap C = C' \cap (N_0 \cap N_1)$. □

C. Run Time Structure for Local Inference

Message JFs created in the last section are used to compute inter-subnet messages. For an agent to reason locally, an inference JT is constructed to process messages from adjacent agents, as specified in Algorithm 9.

Algorithm 9: [BuildInferenceJT] Let A_0 be an agent with local moral graph $G_0 = (N_0, E_0)$. Let adjacent agents be $A_1, ..., A_m$ and F_i be fill-ins that A_0 received from A_i during CommunicateFillin. When A_0 is called to BuildInferenceJT, it does the following:

(1) Eliminate N_0 from $G'_0 = (N_0, E_0 \cup_{i=1}^{m} F_i)$.
(2) Add fill-ins obtained to G'_0.
(3) Construct a JT T_0 from the resultant G'_0.

Figure 6 shows the result of BuildInferenceJT for the example.

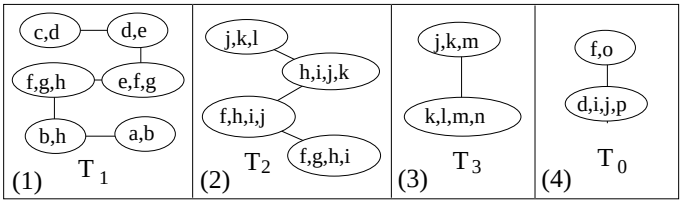

Fig. 6. Inference JTs. (1) The inference JT for agent A_1. (2) The inference JT for A_2. (3) The inference JT for A_3. (4) The inference JT for A_0.

A JT is a special case of JF. We refer to both message JFs and inference JTs as JFs when difference of their roles are unimportant to the presentation.

D. Linking Message JFs and Inference JTs

Message passing during lazy inference, similarly to product-based inference in LJF, consists of a round of inward propagation and a round of outward propagation along the hypertree. Between each pair of adjacent agents, one message is passed in each direction using a given message JF. For example, a message is passed from A_1 to A_2 using the message JF $T_{1\to2}$ and one from A_2 to A_1 using $T_{2\to1}$.

Since a message is obtained from clusters of sending JF and absorbed into clusters of receiving JF, a channel called *linkage*, similar to that in LJF, is created between a sending cluster in a JF and a receiving cluster in another JF. The two clusters are called the *hosts* of the linkage. Unlike its counterpart in LJF, a linkage here is directed. That is, the sending host only has an out-buffer and the receiving host has only an in-buffer. Multiple linkages may exist between a pair of JFs.

As an example, consider linkages from agent A_2 to leaf agent A_1 on the hypertree. The sending JF is message JF $T_{2\to1}$ and receiving JF is inference JT T_1. Figure 7 (a) shows the linkage as a dashed arrow.

Next, consider linkages from A_1 to A_2. The sending JF is message JF $T_{1\to2}$. Since A_2 is an internal agent on the hypertree, it has 3 receiving JFs of $T_{1\to2}$: inference JT T_2, message JFs $T_{2\to0}$ and $T_{2\to3}$. Figure 7 (b) shows the linkages from $T_{1\to2}$ to $T_{2\to3}$. Note that no linkage is connected to the cluster $\{k, l\}$.

We now define linkages in general.

Definition 5: Let A_0 be an agent over N_0 and A_1 be an adjacent agent over N_1. Let $G_{0\to1}$ be the chordal graph at A_0 resultant from CommunicateFillin. Then each clique in the subgraph of $G_{0\to1}$ spanned by $N_0 \cap N_1$ is a `linkage` from A_0 to A_1.

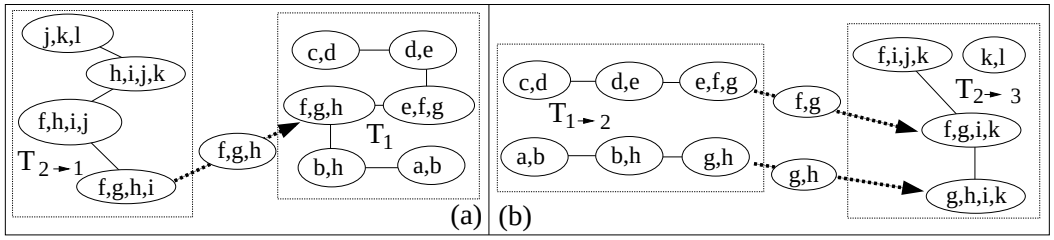

Fig. 7. (a) The linkage from message JF $T_{2\to1}$ at agent A_2 to inference JT T_1 at agent A_1. (b) The linkages from message JF $T_{1\to2}$ at A_1 to message JF $T_{2\to3}$ at A_2.

From BuildMessageJF, linkages from A_0 to A_1 are cliques in $L_{0\to1}$. Next, we define linkage hosts.

Definition 6: Let $T_{0\to1}$ be a message JF from A_0 to A_1, and T_1' be a message JF from A_1 to a third agent or the inference JT of A_1. For each linkage Q from A_0 to A_1, a cluster C in each of the above JF is selected as the `host` of Q if $C \supseteq Q$.

Proposition 5: Linkage host in Definition 6 is well defined.

Proof: From BuildMessageJF, each clique in $L_{0\to1}$ either becomes a cluster in the message JF or is removed due to existence of a superset cluster. Hence, a host exists for this linkage in $T_{0\to1}$.

Next, consider T_1' as a message JF. It is constructed from a chordal graph G' produced by CommunicateFillin and G' contains all fill-ins A_0 sent to A_1. Hence, for each linkage from A_0 to A_1, a host exists in T_1'. The similar argument holds if T_1' is an inference JT. □

Once linkages are determined, the set of message JFs and inference JTs forms a *double-linked junction forest*, where *double* refers to the two message JFs between each pair of adjacent agents.

E. Belief Assignment

At each agent, for each cluster in its inference JT, a set of probability tables from its subnet is assigned with no multiplication performed. Each table in the subnet is assigned to exactly one cluster that contains its domain. Such belief assignment is also performed for each message JF at the agent.

The *joint system belief* of the DLJF is defined as

$$B(\mathcal{N}) = \prod_i \prod_j \prod_k \beta_{i,j,k},$$

where i indexes inference JTs, j indexes clusters in a given JT, $\beta_{i,j}$ denotes the set of potentials (tables) assigned to jth cluster in ith JT, and $\beta_{i,j,k}$ is the kth potential in the set. $B(\mathcal{N})$ is identical to the jpd of the MSBN. Note that if the inference JT at a given agent is replaced by a message JF at the agent, $B(\mathcal{N})$ remains unchanged.

F. Extending Lazy Propagation over Linkages

Lazy inference in DLJFs must process message over linkages, which requires extending operations of Section IV.

To incorporate potentials over linkages, we extend SendPotential by extending the notion of adjacency: Two clusters are *adjacent* if (1) they are directly connected in a JT, or (2) they are hosts of the same linkage. We refer to the extended SendPotential as SendPotential*.

We redefine CollectPotential and DistributePotential to process messages over linkages. They use extended adjacency. In the algorithms, C is a cluster in a JT and `caller` is the local agent or an adjacent cluster not connected through a linkage.

Algorithm 10—CollectPotential*: When CollectPotential* is called in cluster C, C does the following:

(1) For each adjacent cluster Q not connected through a linkage except caller, call CollectPotential* in Q and get incoming potentials from Q.

(2) SendPotential* relative to caller if it is an adjacent cluster.

Note that CollectPotential* only receives messages from linkage in-buffers and does not send to linkage out-buffers because calling CollectPotential* across linkages is disallowed. Under the multiagent paradigm, CollectPotential* is a local operation of an agent, while sending messages across linkages involves a remote agent. CollectPotential* can be executed autonomously to answer local queries, while message passing across linkages requires coordination and incurs communication cost. Next, we redefine DistributePotential as DistributePotential* below. We can then redefine Algorithm 4, using CollectPotential* and DistributePotential*, which we refer to as UnifyPotential*.

Algorithm 11—DistributePotential*: When DistributePotential* is called in cluster C, for each adjacent cluster Q not connected through a linkage except caller, C does the following:

(1) C performs SendPotential* relative to Q.

(2) C calls DistributePotential* in Q.

G. Lazy Inference in DLJF

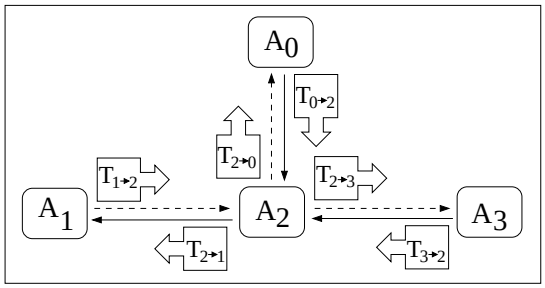

Fig. 8. Lazy communication initiated by agent A_1. The inward message passing is shown by solid arrows using the message JFs associated with the arrows. The outward message passing is shown by dotted arrows using a different set of message JFs.

Lazy inference with DLJF consists of lazy communication among agents using message JFs followed by local lazy propagation at each agent using its inference JT. Figure 8 illustrates lazy communication initiated by agent A_1. The first round of inward message passing is shown by solid arrows using the message JFs associated with the arrows. The second round of outward message passing is shown by dotted arrows using a different set of message JFs. Algorithm 12 is used recursively by agents for inward lazy communication.

Algorithm 12—CollectBeliefLDLJF: When A_c calls A_0 to CollectBeliefLDLJF, A_0 does the following:

(1) For each adjacent agent A_i except A_c, A_0 calls CollectBeliefLDLJF in A_i and gets message from A_i.

(2) If A_c is an adjacent agent, A_0 calls CollectPotential* in each linkage host of $T_{0\rightarrow c}$, followed by SendPotential* along the linkage.

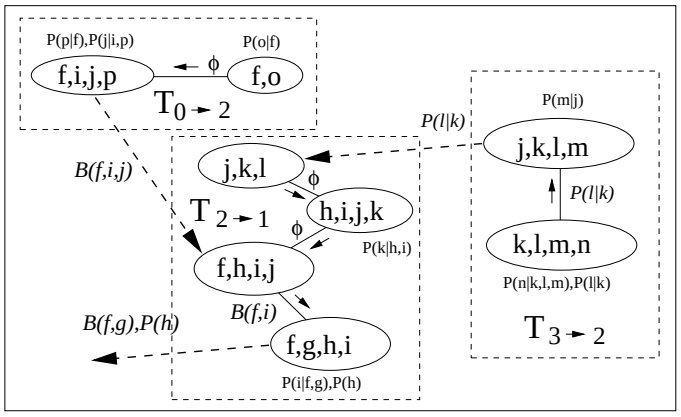

Fig. 9. Inward lazy communication CollectBeliefLDLJF initiated by A_1. The three message JFs involved are shown. Beside each cluster is the set of potentials assigned to it. Solid arrows indicate intra-subnet messages and dashed arrows show inter-subnet messages. Message outgoing from $T_{2\rightarrow 1}$ goes to A_1. $B()$ denotes a newly produced potential.

Figure 9 illustrates CollectBeliefLDLJF initiated by A_1. Algorithm 13 performs outward lazy communication on the hypertree.

Algorithm 13—DistributeBeliefLDLJF: When A_c calls A_0 to DistributeBeliefLDLJF, for each adjacent agent A_i except A_c, A_0 performs the following:

(1) Call CollectPotential* in each linkage host of $T_{0\rightarrow i}$, followed by executing SendPotential* along the linkage.

(2) Call DistributeBeliefLDLJF in A_i.

Algorithm 14 combines CollectBeliefLDLJF and DistributeBeliefLDLJF for lazy communication. It is executed by the system coordinator.

Algorithm 14—CommunicateBeliefLDLJF:

(1) Select an agent A arbitrarily.

(2) Call CollectBeliefLDLJF in A.

(3) Call DistributeBeliefLDLJF in A.

Theorem 2 establishes that CommunicateBeliefLDLJF is exact.

Theorem 2: For each agent A_i over N_i, after CommunicateBeliefLDLJF, its inference JT T_i satisfies the following, where j indexes inference JTs:

$$B_{T_i}(N_i) = \sum_{\mathcal{N}\setminus N_i} [\prod_j B_{T_j}(N_j)]$$

Proof: The hypertree of MSBN is isomorphic to a JT Υ over $\mathcal{N}$, where each cluster is the subdomain N_i. Let Θ_i denote the set of potentials assigned to inference JT T_i. We associate each cluster N_i of Υ with Θ_i. Hence we have

$$\prod_{\theta\in\Theta_i} \theta = B_{T_i}(N_i),$$

where $B_{T_i}(N_i)$ denotes the assigned belief of agent A_i. Due to equivalent belief assignment to inference JT and message JFs, the equation also holds for each message JF $B_{T_{i\rightarrow j}}(N_i)$ where j indexes an adjacent agent of A_i. Hence, the joint system belief of Υ is identical to that of the DLJF.

Suppose lazy propagation is performed in Υ. Let $\Theta_{i\rightarrow j}$ denote the message that cluster N_i sends to an adjacent cluster N_j. Potentials in $\Theta_{i\rightarrow j}$ are dependent on the order in which marginalization is performed during lazy propagation. However, any instance of $\Theta_{i\rightarrow j}$ is equivalent to any other, as the product of its potentials is identical.

Now it suffices to show that there exists a marginalization order such that the resultant $\Theta_{i\rightarrow j}$ is identical to the set of potentials sent from $T_{i\rightarrow j}$ to A_j during CommunicateBeliefLDLJF. In other words, the union of potentials over all linkages from $T_{i\rightarrow j}$ to A_j is identical to the message $\Theta_{i\rightarrow j}$ during lazy propagation in Υ.

The message from $T_{i\rightarrow j}$ to A_j is generated through a sequence of marginalizations guided by the message JF. Let O be the order used. Because $T_{i\rightarrow j}$ is created

from chordal graph obtained in cooperative triangulation, O must be consistent with a node elimination order that can lead to construction of $T_{i\rightarrow j}$. Due to the parallel between node elimination and marginalization, the same order can be applied to marginalization during lazy propagation in Υ. The resultant $\Theta_{i\rightarrow j}$ will be identical to the set of potentials sent by $T_{i\rightarrow j}$ to A_j during CommunicateBeliefLDLJF. □

Note that after CommunicateBeliefLDLJF terminates, an agent can call UnifyPotential* in a cluster in its inference JT. Then, exact marginal probability distribution over each cluster will be be available at the cluster.

H. Complexity Analysis

We use the following parameters:

- n: the total number of agents.
- c: the maximum number of clusters in an inference JT or message JF.
- r: the maximum number of linkages between a pair of message JFs.
- q: the cardinality of the largest cluster.
- f: the cardinality of the largest family.
- m: the total number of variables.

First, consider the time complexity of UnifyPotential*: Each cluster sends a message to and receives a message from each adjacent cluster. Hence, the complexity is linear in c. In the simplest case, the message is empty and its computation is trivial. In the most complex case, the potential over the cluster needs to be obtained by product which is then marginalized into the potential over the separator. Therefore, the time complexity of UnifyPotential* is between O(2c) and $O(2c\ 2^q)$.

Next, consider the time complexity of lazy communication: A total of $2(n-1)$ inter-subnet messages are sent during CommunicateBeliefLLJF with two along each hyperlink. Each message requires a round of inward lazy propagation in a message JF of the complexity between O(c) and $O(c\ 2^q)$. After CommunicateBeliefLLJF, each agent needs to execute UnifyPotential* to obtain cluster marginals. Hence, time complexity is between $O(4n\ c)$ and $O(4n\ c\ 2^q)$.

For space complexity, each agent maintains an inference JT and as many message JFs as the number of adjacent agents. A total of n JTs and $2(n-1)$ message JFs are maintained. In the simplest case, one copy of conditional probability tables is maintained and no new potential is generated during inference, which results in the space complexity of $O(m\ 2^f)$. In the most complex case, cluster potentials are multiplied in each cluster of each JF during inference, which yields the overall space complexity of $O(3n\ c\ 2^q + m\ 2^f)$.

The above result can be compared with that of product-based inference with LJF. Local inference in each agent has time complexity of $O(2c\ 2^q)$. Two rounds of such inference is needed during agent communication, hence the time complexity of $O(2n\ c\ 2^q)$. To store a total of n JTs and $2(n-1)$ linkage trees, the space complexity is $O(n\ (c+2\ r)\ 2^q)$. Both time and space complexity of product-based inference with LJF are much higher than the lower bound result of lazy inference with DLJF.

VI. Lazy Inference with Linked Junction Forest

Lazy inference with DLJF is more efficient than product-based inference with LJF as shown by the above analysis. However, agents maintain a total of $2(n-1)$ message JFs (each over a subdomain), while agents with LJF maintain $2(n-1)$ linkage trees (each over a d-sepset, which is generally much smaller than the subdomain). We present an alternative method for lazy inference with LJF that explores this opportunity for further space savings. It uses the same runtime structure as product-based inference, except that potentials assigned to each cluster are not multiplied. It turns out although lazy inference with DLJF uses only marginalization and multiplication of potentials, as lazy propagation does, the alternative method also requires division as defined below:

Algorithm 15—Lazy Division: Let α and γ be two sets of potentials. The `lazy division` of α by γ, denoted $\alpha/_L\gamma$ is performed as follows:

1) If a potential appears in both α and γ, delete it from both.
2) For each potential f in γ, delete f from γ, multiply the set θ of potentials in α whose domains overlap with that of f, divide the product by f, and replace θ in α by the result of the division.

For example, let $\alpha = \{P(a), P(b|a), P(c|b), P(d|c)\}$ and $\gamma = \{P(a), P(b)\}$. Then $\alpha/_L\gamma = \{P(b|a) * P(c|b)/P(b), P(d|c)\}$. Note that the product of potentials in $\alpha/_L\gamma$ is identical to the product of potentials in α divided by the product of potentials in γ.

A. Lazy Communication in LJF

Communication also consists of an inward round and an outward round of message passing on the hypertree. Figure 10 illustrates inward propagation in the LJF of Figure 2, initiated by agent A_0.

First, UnifyPotential* is performed by A_1 and A_2. Consider A_1. At linkage host $\{b, c, f\}$, potentials over linkage $\{b, c\}$ is computed from local potentials plus potentials from cluster $\{b, c, h\}$. The resultant is $B(b, c)$. At linkage host $\{a, b, e\}$, potentials $P(a)$ and $B(b)$ over linkage

$\{a, b\}$ are computed. As a result, both linkages in $L_{0,1}$ pass information on b: a duplication. To remove the duplication, A_1 examines potentials at linkage $\{a, b\}$ and identifies $B(b)$ as duplicated information. After $B(b)$ is deleted, potentials from $L_{0,1}$ to T_0 become $B(b, c)$ over linkage $\{b, c\}$ and $P(a)$ over linkage $\{a, b\}$. After a similar operation at A_2, inward communication ends.

Outward communication follows, during which A_0 sends messages to A_1 and A_2. To obtain message to A_1, A_0 performs UnifyPotential* using linkage potentials from A_2 but not those from A_1. To compute message to A_2, A_0 performs another UnifyPotential* using linkage potentials from A_1 but not those from A_2. Again, duplicated information on variable b needs removed. This ends lazy communication.

During communication, potentials are sent from one agent with JT T to an adjacent agent with JT T' through the linkage tree. The potentials are obtained from linkage hosts in T. To ensure that each linkage host has the necessary information, UnifyPotential* must be performed before these potentials are computed. This renders T locally consistent. As a result, for every two linkages adjacent in linkage tree, same information on their shared variables are sent by their hosts. If such potentials are directly passed to T', the new belief in T' will be incorrect due to duplication. We present below in general how to use lazy division to compute cross-linkage potentials without duplication.

First, the linkage tree L from T to T' is directed. For each linkage Q in L, the following message buffers are created.

- in-buffer$_1$: in-buffer from host cluster in T.
- in-buffer$_2$: in-buffer from parent linkage in L. If Q has no parent linkage, its in-buffer$_2$ is null.
- out-buffer$_1$: out-buffer to host cluster in T'.
- out-buffer$_2$, out-buffer$_3$, ... : out-buffers to child linkages in L.

Potentials from Q to T' are computed as follows:

Algorithm 16—SendLinkageMsg: For each linkage Q, Q requests its linkage host to fill in-buffer$_1$ by SendPotential* relative to Q. After both in-buffers are filled, Q does the following:

(1) For each child linkage Q', marginalize out variables $Q \setminus Q'$ from potentials in in-buffer$_1$, and send resultant potentials to the out-buffer to Q'.

(2) Divide the set α of potentials in in-buffer$_1$ by the set γ of potentials in in-buffer$_2$ with lazy division and send $\alpha/_L \gamma$ to out-buffer$_1$.

Note that sending to out-buffer$_1$ involves inter-agent message passing. Using SendLinkageMsg, algorithms below perform lazy communication in LJF. CollectBeliefLLJF defines inward communication.

Algorithm 17—CollectBeliefLLJF: When CollectBeliefLLJF is called in agent A, A does the following:

(1) If caller is not the only adjacent agent, call CollectBeliefLLJF in each adjacent agent except caller. After all calls are completed, receive linkage potentials from each adjacent agent except caller.

(2) If caller is an adjacent agent, do UnifyPotential* using linkage potentials from each adjacent agent except caller, followed by SendLinkageMsg relative to caller.

DistributeBeliefLLJF defines outward lazy communication.

Algorithm 18—DistributeBeliefLLJF: When DistributeBeliefLLJF is called in A, for each adjacent agent A' except caller, A does the following:

(1) A does UnifyPotential* using linkage potentials from each adjacent agent except A'.

(2) A does SendLinkageMsg relative to A'.

(3) A calls DistributeBeliefLLJF in A'.

CommunicateBeliefLLJF combines the above algorithms for lazy inference with LJF and is executed by the system coordinator.

Algorithm 19—CommunicateBeliefLLJF:

(1) Select an agent A arbitrarily.

(2) Call CollectBeliefLLJF in A.

(3) Call DistributeBeliefLLJF in A.

An agent A calls UnifyPotential* before sending messages to each adjacent agent. If A has k adjacent agents, then one call is made during CollectBeliefLLJF, $k-1$ calls are made during DistributeBeliefLLJF. Hence, a total of k rounds of local lazy propagations are needed during CommunicateBeliefLLJF.

B. Soundness

We use $const$ to denote a positive constant. Proposition 6 shows that message sent over a linkage tree defines marginal potential over d-sepset.

Proposition 6: Let T over N be a local JT, T' be a local JT adjacent to T, I be their d-sepset, and L be the linkage tree over I. Let UnifyPotential* be performed in T followed by SendLinkageMsg relative to T'. Let $B(N)$ be the potential $B(N) = \prod_{C \in T} \beta(C) \prod_{Q' \notin L} \beta(Q')$, where $\beta(C)$ is the product of potentials assigned to a cluster C, $\beta(Q')$ is the product of potentials received from a linkage Q', and only linkages other than those in L are included. For each linkage $Q \in L$, let $\alpha(Q)$ be the product of potentials that Q sends to T' by SendLinkageMsg. Then

$$\prod_{Q \in L} \alpha(Q) = const \sum_{N \setminus I} B(N).$$

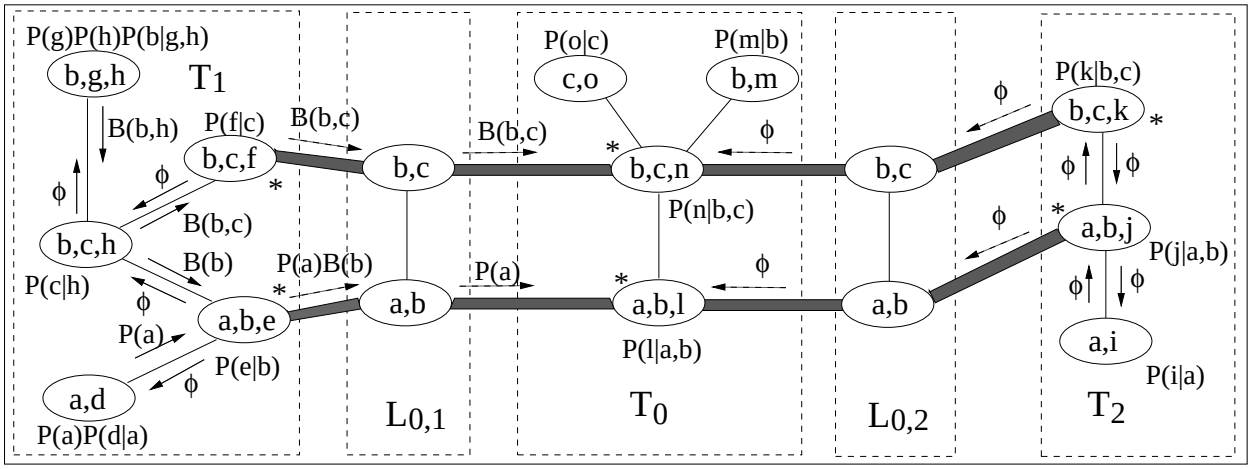

Fig. 10. Inward communication in the LJF of Figure 2, initiated by agent A_0.

Proof: First, we consider effect of UnifyPotential* using Proposition 1. To do so, for each cluster C in T, we define equivalent cluster potential $\beta'(C) = \beta(C) \prod_{Q' \to C} \beta(Q')$, where $Q' \to C$ indexes linkage Q' that feeds message to C. We can then ignore such Q' in the remaining proof and Proposition 1 is applicable.

Next, for any linkage $Q \in L$, consider its host X. From Proposition 1, after UnifyPotential*, the set of potentials (including those from its in-buffers) associated with X defines the marginal of $B(N)$ onto X. This set, marginalized onto Q, is sent to in-buffer$_1$ of Q. Denote the product of potentials in in-buffer$_1$ by $\alpha'(Q)$ and the product of potentials in in-buffer$_2$ by $\theta'(Z)$, where Z is the separator between Q and its parent linkage. Since a linkage tree is a JT,

$$\prod_{Q \in L} \alpha'(Q) / \prod_{Q \in L} \theta'(Z) = const \sum_{N \setminus I} B(N).$$

The proposition follows because the message that Q sends to out-buffer$_1$ is obtained by lazy division, which is equivalent to $\alpha(Q) = \alpha'(Q)/\theta'(Z)$. □

Theorem 3 shows that local potentials of an agent and linkage tree messages it receives define the marginal of joint system potential:

Theorem 3: Let F over $\mathcal{N}$ be LJF of an MSBN with joint system potential $B_F(\mathcal{N})$ and CommunicateBeliefLLJF be performed in F. Let T be any local JT over N and $B(N)$ denote potential $\prod_{C \in T} \beta(C) \prod_{Q \to T} \beta(Q)$, where $\beta(C)$ is product of potentials assigned to cluster C, $\beta(Q)$ is product of potentials received from linkage Q into T (denoted by $Q \to T$). Then, $B(N) = const \sum_{\mathcal{N} \setminus N} B_F(\mathcal{N})$.

Proof: Let A denote the agent associated with T. Direct the hypertree of F with A as the root. During CommunicateBeliefLLJF, only inter-subnet messages directed towards A has an impact on $B(N)$. These messages are sent in semi-parallel order from leaves to root. We analyze the impact of these messages by making agents send one by one starting from any leaf agent A'.

Since A' (over subdomain N') is a leaf, it is adjacent to one agent A'' (over subdomain N''). By Proposition 6, message A' sent to A'' defines marginal of $B(N')$ onto their d-sepset. Since the message is the only impact that A' has on $B(N)$ and it is received by A'', agent A' is effectively removed from the system. The new joint system potential defined by the local potentials in remaining agents and messages A'' received is $const \sum_{N' \setminus N''} B_F(\mathcal{N})$.

By applying the above argument recursively to each leaf agent, eventually, all other agents in F will be removed except A. The result follows. □

Corollary 1 states that local potentials of a cluster and its in-buffer potentials define marginal of joint system potential. Here, in-buffers include those from adjacent clusters in the same JT and those from linkages.

Corollary 1: Let F over $\mathcal{N}$ be LJF of an MSBN with joint system potential $B_F(\mathcal{N})$ and let CommunicateBeliefLLJF be performed in F. Let A be any agent that performs UnifyPotential*, after termination of CommunicateBeliefLLJF, using linkage potentials from its adjacent agents. Let C be any cluster in the local JT of A and $B(C)$ be the potential

$$B(C) = \beta(C) \prod_{R \to C} \beta(R) \prod_{Q \to C} \beta(Q),$$

where $\beta(C)$ is product of potentials assigned to C, $\beta(R)$ is product of potentials received from in-buffer associated with a separator R, and $\beta(Q)$ is product of potentials received from a linkage Q.

Then, $B(C) = const \sum_{\mathcal{N} \setminus C} B_F(\mathcal{N})$.

Proof: It follows from Theorem 3 and Proposition 1. Theorem 3 ensures the marginal of $B_F(\mathcal{N})$ onto the subdo-

TABLE I
PERFORMANCE COMPARISON

	Digital System MSBN (a)			Simulated MSBN (b)			Simulated MSBN (c)		
	PLJF	LDLJF	LLJF	PLJF	LDLJF	LLJF	PLJF	LDLJF	LLJF
Comm. Time (s)	1.6	2.0	1.5	4.0	2.55	2.0	128.4	16.9	127.0
Mem. Usage (kb)	1150	1213	1160	1352	1320	1272	7693	3553	2698
Data/Code Mem. (kb)	190	253	200	392	360	310	6733	2593	1738

main of the local JT and Proposition 1 ensures further marginalization onto C. □

C. Complexity Analysis

First, consider the time complexity: During CommunicateBeliefLLJF, a total of $2(n-1)$ SendLinkageMsg is performed twice along each hyperlink. Before each performance, UnifyPotential* is executed. After CommunicateBeliefLLJF, UnifyPotential* may be executed by each agent. Since communication is dominated by the computation of UnifyPotential*, the time complexity is between $O(4n\ c)$ and $O(4n\ c\ 2^q)$. It is comparable with that of lazy inference with DLJF.

For space complexity, each agent maintains an inference JT and as many linkage trees as the number of adjacent agents. A total of n JTs and $2(n-1)$ linkage trees are maintained. In the simplest case, one copy of conditional probability tables is maintained and no new potential is generated during inference, which results in the space complexity of $O(m\ 2^f)$. In the most complex case, cluster potentials are multiplied during inference, which produces the overall space complexity $O(n\ (c+2r)\ 2^q + m\ 2^f)$. The space complexity is lower than that of lazy inference with DLJF.

VII. EXPERIMENTAL COMPARISON

As the complexity analysis for each lazy inference method (Sections V-H and VI-C) can only provide widely separated complexity bounds, it is informative to compare their performance experimentally.

All three inference methods are implemented in WebWeavr [3] in Java. We report the experimental results using three MSBNs. The first (a) is the digital system MSBN in [13] with a domain size of 91 variables, the second (b) is a simulated MSBN [15] with a domain size of 201, and the third (c) is also simulated with a domain size of 998. Figure 11 shows the subdomain profiles of the three MSBNs.

The five agents are run on 4 computers connected through a local network: one Sun-Blade-1000 station with 750MHz Ultra-SPARC-3 processor, two HP-X2100 workstation with 2.4GHz Pentium-4 processor, and one Acer-Travel-Mate-630 Laptop with 1.4GHz Pentium-4 Mobile CPU.

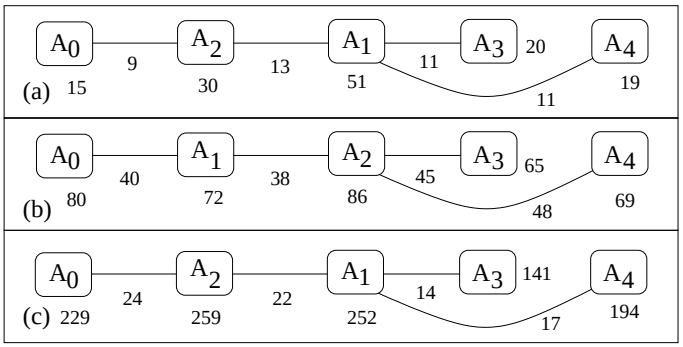

Fig. 11. Profiles of MSBNs: (a) digital system,(b) simulated and (c) simulated. Beside each agent is the number of variables contained in its subdomain. Each hyperlink is labeled with the size of the d-sepset.

For each inference method, we record down the total communication time of the multiagent system and memory usage of the agent with the largest subdomain. Table I shows the experimental data, where PLJF refers to product-based inference with LJF, LDLJF refers to lazy inference with DLJF, and LLJF refers to lazy inference with LJF. All memory usage includes roughly about 960 kb Java virtual machine (JVM) and GUI related classes. As an example, for the digital system agent with the largest subdomain, its subnet data and additional code take 1150 - 960 = 190 kb under PLJF.

For digital system MSBN, LDLJF takes longer time and uses more memory than PLJF. LLJF is slightly faster than PLJF and also uses more memory than PLJF. This result seems surprising. Our analysis is the following: LDLJF and LLJF take more code memory than PLJF due to more sophisticated control in lazy inference [1] than in product-based inference. LDLJF needs more data memory than LLJF as a message JF (over subdomain) takes

[1] In lazy inference, it is necessary to determine whether marginalization relative to a variable can be performed by a trivial deletion of a potential or by multiplication plus marginalization. To allow such decisions to be made effectively, the head and tail of a potential needs to be maintained. The control to make such decisions is not needed in product-based inference.

more memory than a linkage tree (over d-sepset). The advantage of factorization in general should reduce data memory in lazy inference. However, the advantage is not sufficiently high in the digital system MSBN since the domain is small and sparse (the largest cluster of LJF has size 6). This analysis also explains the comparison in communication time.

The comparison changes in the simulated MSBN (b), where LDLJF and LLJF are faster than PLJF and use less memory. LDLJF uses 25% less time and LLJF uses 50% less time than PLJF. Without counting the memory used by JVM and GUI, LDLJF uses 8% less memory and LLJF uses 21% less memory than PLJF. This is because the simulated MSBN has a much larger domain and is less sparse (the largest cluster has size 9). The advantage of factorization in lazy inference has overridden the negative effect of extra code.

Experiment with the simulated MSBN (c) confirms the same pattern but with more significant performance difference. LDLJF uses 39% of memory compared to PLJF and LLJF uses only 26%. Time-wise, LLJF and PLJF performed at the same level. However, LDLJF used only 13% of the time used by PLJF: 7.6 times faster. We attribute the significant speed up to the direction-dependent triangulation with LDLJF which can produce more sparse runtime structure than that of LLJF. The results demonstrate that as the domain size further increases and dependence structure becomes denser, more significant computational savings can be expected for LDLJF and LLJF.

VIII. Conclusion

We extend lazy propagation for single-agent inference in BNs to lazy inference in multiagent MSBNs. We presented two methods, one based on runtime DLJF and another on LJF. Lazy inference with DLJF employs message direction dependent triangulation. It produces sparser triangulation and uses only multiplication and marginalization in inference. However, each agent needs to maintain a message JF for each adjacent agent. Lazy inference with LJF uses direction independent triangulation. Less sparse triangulation may be produced and lazy division is needed during inference. However, each agent needs only to maintain a linkage tree for each adjacent agent. These methods, like product-based method with LJF, are exact[2]. When the domain structure is sparse, they are efficient.

[2]Our analysis of soundness only considered communication of prior belief. Since effect of an observation on variable x is equivalent to multiplying potential $P(x|\pi(x))$ by an observation function $f(x)$, our analysis can be extended to posterior in a straightforward way. To keep the paper concise, we have chosen not to elaborate.

Both new methods use more sophisticated control than product-based inference and hence take more memory space for code. If the domain is not sufficiently large and the dependence structure is not sufficiently dense, the advantage of new methods over product-based method is limited. However, when the domain is large and especially dependence structures are reasonably dense, new methods gain efficiency in both space and time. Hence, the new methods allow multiagent uncertain reasoning to be performed in much larger domains given the computational resource.

Acknowledgement

Financial support from NSERC, Canada to the first author is acknowledged.

References

[1] E. Castillo, J. Gutierrez, and A. Hadi. *Expert Systems and Probabilistic Network Models*. Springer, 1997.

[2] R.G. Cowell, A.P. Dawid, S.L. Lauritzen, and D.J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[3] P. Haddawy. An overview of some recent developments in Bayesian problem-solving techniques. *AI Magazine*, 20(2):11–19, 1999.

[4] F.V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, New York, 2001.

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

[6] A.L. Madsen and F.V. Jensen. Lazy propagation in junction trees. In *Proc. 14th Conf. on Uncertainty in Artificial Intelligence*, 1998.

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

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

[9] D.J. Rose, R.E. Tarjan, and G.S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Computing*, 5:266–283, 1976.

[10] G. Shafer. *Probabilistic Expert Systems*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[11] Y. Xiang. Belief updating in multiply sectioned Bayesian networks without repeated local propagations. *Inter. J. Approximate Reasoning*, 23:1–21, 2000.

[12] Y. Xiang. Cooperative triangulation in MSBNs without revealing subnet structures. *Networks*, 37(1):53–65, 2001.

[13] Y. Xiang. *Probabilistic Reasoning in Multi-Agent Systems: A Graphical Models Approach*. Cambridge University Press, 2002.

[14] Y. Xiang. Comparison of multiagent inference methods in multiply sectioned Bayesian networks. *Inter. J. on Approximate Reasoning*, 33(3):235–254, 2003.

[15] Y. Xiang, X. An, and N. Cercone. Simulation of graphical models for multiagent probabilistic inference. *Simulation: Trans. Society for Modeling and Simulation*, 79(10):545–567, 2003.

[16] Y. Xiang and X. Chen. Inference in multiply sectioned Bayesian networks with lazy propagation and linked jounction forests. In *Procs. 2nd European Workshop on Probabilistic Graphical Models*, pages 217–224, 2004.

[17] Y. Xiang and F.V. Jensen. Inference in multiply sectioned Bayesian networks with extended Shafer-Shenoy and lazy propagation. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pages 680–687, Stockholm, 1999.

[18] Y. Xiang and V. Lesser. On the role of multiply sectioned Bayesian networks to cooperative multiagent systems. *IEEE Trans. Systems, Man, and Cybernetics-Part A*, 33(4):489–501, 2003.

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

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