Fault-Tolerant Multiagent Exact Belief Propagation

Xiangdong An and Nick Cercone
Department of Computer Science and Engineering
York University
Toronto, ON M3J 1P3, Canada
xan@cs.yorku.ca, ncercone@yorku.ca

Abstract
Multiply sectioned Bayesian networks (MSBNs) support multiagent probabilistic inference in distributed large problem domains, where agents (subdomains) are organized by a tree structure (called hypertree). In earlier work, all belief updating methods on a hypertree are made of two rounds of propagation, each of which is implemented as a recursive process. Both processes need to be started from the same designated (root) hypernode. Agents perform local belief updating at most in a partial parallel manner. Such methods may not be suitable for practical multiagent environments since they are easy to crush for the problems happened in communication or local belief updating. In this paper, we present a fault-tolerant belief updating method for multiagent probabilistic inference. In this method, multiple agents concurrently perform exact belief updating in a complete parallel. Temporary problems happened from time to time at some agents or some communication channels would not prevent agents from eventually converging to the correct beliefs. Permanently disconnected communication channels would not keep the properly connected portions of the system from appropriately finishing their belief updating within portions. Compared to the previous traversal-based belief updating, the proposed approach is not only fault-tolerant but also robust and scalable.

Keywords: multiagent probabilistic reasoning, Bayesian networks, fault-tolerant computing, distributed computing, parallel computing

1 Introduction
Multiply sectioned Bayesian networks (MSBNs) (Xiang, 2002) provide a coherent framework for probabilistic inference in distributed multiagent interpretation systems. They support object-oriented inference (Koller & Pfeffer, 1997) and have been applied in many areas such as medical diagnosis (Xiang, Pant, Eisen, Beddoes, & Poole, 1993a), equipment monitoring and diagnosis (Xiang & Geng, 1999), distributed network intrusion detection (Ghosh & Sen, 2004), and distributed collaborative design (Xiang, Chen, & Havens, 2005).

1.1 MSBN Belief Propagation Architectures
In an MSBN, all Bayesian subnets (and the corresponding agents) are organized into a tree structure called hypertree. Adjacent agents on a hypertree directly communicate with each other via their
shared variables, called their interface. Inference with MSBNs can be done with their compiled representations called linked junction forests (LJFs) (Xiang, Poole, & Beddoes, 1993b). A LJF is a collection of junction trees (JTs) (Jensen, Lauritzen, & Olesen, 1990), where each JT corresponds to a Bayesian subnet. There is also a JT corresponding to each interface, called linkage tree. A clique in a linkage tree is called a linkage. A comparison (Xiang, 2003) between LJF-based belief propagation extended from Hugin architecture (Jensen et al., 1990) and extensions of other inference methods for single-agent Bayesian networks (in particular, the loop-cutset methods and two stochastic sampling methods) indicates that the LJF based inference is superior than those alternatives.

For belief updating extended from Hugin architecture, all interface nodes should have consistent linkage tree structures at different agents (Xiang, 2002; Zhang, Tian, & Lu, 2001), which can be achieved by cooperative multiagent triangulation of the consistently moralized MSBNs (Xiang, 2001). For belief propagation extended from Shenoy-Shafer (Shenoy & Shafer, 1986) or Lazy (Madsen & Jensen, 1999) architectures, direction-dependent junction forests (JFs) are allowed for message passing between adjacent agents, where the direction-dependent JFs for inter-agent message passing and the JT's for intra-agent belief updating are collectively called a Doubly-linked junction forest (DLJF) (Xiang & Jensen, 1999; Xiang, Jensen, & Chen, 2006). DLJFs can be obtained via direction dependent triangulation (Xiang et al., 2006). On large and dense problem domains, the propagation extended from Lazy architecture (or Shenoy-Shafer architecture) based on DLJFs is more efficient but may take more memory space than that based on LJFs with consistent linkage trees (which we simply refer to as LJFs), though both gain efficiency in space and time over the belief propagation extended from Hugin.

Algorithm 1 (CommunicateBelief) (Xiang, 2002) An LJF representation is populated by multiple agents with one at each hypernode. The system coordinator does the following: Choose an agent A* arbitrarily. Call CollectBelief on A* followed by a call of DistributeBelief on A*.

Algorithm 2 (CollectBelief) (Xiang, 2002) Let A_0 be an agent with local JT T_0. A caller is either an adjacent agent A_c or the system coordinator. Denote additional adjacent agents of A_0 by A_1, ..., A_m if any. When the caller calls on A_0 to CollectBelief, it does the following:
1. If A_0 has no adjacent agent except caller, it performs UnifyBelief at T_0 and returns.
2. Otherwise, for each agent A_i (i = 1, ..., m), A_0 calls CollectBelief on A_i. After A_i finishes, A_0 calls on itself to UpdateBelief relative to A_i.

Algorithm 3 (DistributeBelief) (Xiang, 2002) Let A_0 be an agent with local JT T_0. A caller is either an adjacent agent A_c or the system coordinator. Denote additional adjacent agents of A_0 by A_1, ..., A_m if any. When the caller calls on A_0 to CollectBelief, it does the following:
1. If caller is an adjacent agent, A_0 calls UpdateBelief on itself relative to caller.
2. For each agent A_i (i = 1, ..., m), A_0 calls DistributeBelief on A_i.

So far, all LJF or DLJF based belief propagation methods (extended from Hugin, Shenoy-Shafer, or Lazy architecture), exact (Xiang, 2000; Xiang & Jensen, 1999; Xiang et al., 2006) or approximate (Zhang et al., 2001), were similarly presented as a set of three algorithms: a main algorithm and two recursive algorithms as shown by Algorithms 1, 2, and 3. The main algorithm (CommunicateBelief) arbitrarily selects a hypernode as the root hypernode and activates the two recursive algorithms at the hypernode in a sequence. The two recursive algorithms are generally called the collection process (CollectBelief) and the distribution process (DistributeBelief), respectively. The distribution process
is started only after the collection process finishes. The two recursive processes go through all hypernodes in a depth-first traversal of the hypertree. At this stage, we do not need to care about the details of UnifyBelief and UpdateBelief called by one or both recursive processes. UnifyBelief simply performs belief updating at a subnet based on local evidence observed and messages received from other agents. UpdateBelief involves belief passing between agents besides belief updating with UnifyBelief at an agent. All existing MSBN belief updating methods perform belief updating based on the three algorithms and they differ only at UnifyBelief and UpdateBelief regarding the specific belief representation, storage, and updating.

1.2 Problems in Adverse Environments

All existing MSBN belief propagation methods based on the three algorithms are too fragile to work in practical multiagent environments. The belief propagation process would crush for any errors happened in the belief updating or communication. Note the two recursive MSBN belief propagation processes (CollectBelief and DistributeBelief) can be implemented as one iterative process, where a hypernode can send a message to a neighbor only after it has received the needed messages from its neighbors, and a hypernode can only send one message to each neighbor in the whole process. However, no matter these methods are implemented based on the two recursive processes or one iterative process, the similar problems would persist in the adverse environments.

![Figure 1: Illustration of recursive (traversal-based) belief propagation on a hypertree.](image)

Figure 1 shows how the control moves among hypernodes (subnets, agents) in one recursive process, where each ellipse denotes a hypernode, the white headed arrows denote the forward direction and the dark headed arrows returning direction. Belief updating messages are passed in forward (DistributeBelief) or backward (CollectBelief) directions. The process as shown in Figure 1 is started at hypernode 0. A possible serial order for agent activation and belief updating is agents 0, 1, 2, 3, 4, 5, and 6 (corresponding to hypernodes 0, 1, 2, 3, 4, 5, and 6, respectively). The process can be made partially parallel. For example, operations at hypernode 1 can be done parallelly with operations at hypernodes 2 through 6. Anyway, for the collection process, operations at hypernode 0 would not finish before operations at all other hypernodes finish since hypernode 0 needs messages from those hypernodes. Similarly, operations at hypernode 5 would not finish before it receives messages from hypernode 6. For the distribution process, operations at hypernode 5 would not start before belief updating operations at hypernode 2 finishes. Note even the problems in the other direction where no updating messages are delivered would crush the whole belief updating process. For example, in the collection process, if agent 0 fails to start agent 2, belief updating on the hypertree would freeze. This indicates any message passing failures on the hypertree would crush the belief updating. If any messages are passed with errors, agents’ beliefs would become inconsistent with each other and there are no mechanisms to correct the problems. Here, we say two agents are inconsistent on their beliefs if they have different beliefs (probability distributions) on their shared variables.
1.3 The Proposed Approach

In this paper, we present an iterated multiagent belief updating approach, where multiple agents perform belief updating iteratively. Here, by “iterated” or “iterative” we mean repeated belief updating and message passing at each agent, which is different from the “iterative” mentioned above regarding the alternative implementation of the recursive processes since in that case an agent can only send one message to a neighbor and this happens only after it receives the messages from all other neighbors. In the following discussion, except in the related work section explicitly noted with italic type, we always use “iterative” to mean the repetitive belief updating and message passing at each agent.

By the proposed approach, agents perform exact belief updating in a complete parallel. Any temporary problems in the belief propagation could delay, but would not prevent the convergence of the agents’ beliefs to the right ones (as if there were no problems). The disconnected communication channels could break a system into several portions. Agents in each connected portion would continue their belief updating until the respective local consistency is reached. When the communication among these portions is recovered, the belief updating among these portions would be resumed until the global consistency is achieved. Any wrong beliefs absorbed would be washed out and be replaced by the correct ones delivered later. This method not only improves the agents’ autonomy and concurrency, but is also fault-tolerant, robust and adaptive. A system based on the iterated belief updating is also open-ended: multiple systems can be combined seamlessly.

The idea is first inspired by the working mechanism of the Pearl’s polytree belief updating algorithm (Pearl, 1988). On a polytree, each node (representing a variable) sends one message to each of its parents and children whenever it receives the messages from all others. The messages sent out and its updated belief completely depend on its own initial belief and the messages received from its parents and children. In the whole process, a node can only send one message to each of its neighbors. Therefore, in the beginning only a leaf node can send the message to its parent or child. The polytree algorithm is subject to the faults in the belief updating because a node cannot send a message to a neighbor before it receives the messages from all other neighbors, and any wrong information received would be desperately propagated to all scheduled nodes. However, if each node keeps sending messages to its neighbors based on its own initial belief and the messages received (before any message is received from a neighbor, the message from the neighbor is assumed to be a uniform distribution), the messages received by each node are guaranteed to converge to the correct ones since the correct messages would expand (propagate) from the leaf nodes to the center of the hypertree. Such belief propagation would also be robust.

Nevertheless, the idea cannot be similarly extended to the Hugin belief propagation with JTs or the extended Hugin belief propagation with LJFs due to their specific belief updating and message passing mechanisms. It turns out the extended Shenoy-Shafer belief propagation and the extended Lazy belief propagation with LJFs and DLJFs have the similar properties as the original polytree algorithm. In this paper, we first investigate why the extended Hugin belief propagation with LJFs could not be iterated, then we investigate how the extended Shenoy-Shafer and Lazy belief propagations can be properly iterated. Detailed facilities and operations, and multiagent and multithread deadlock prevention are discussed.

The rest of the paper is organized as follows. In Section 2, the related work is reviewed. An overview of MSBNs is given in Section 3. In Section 4, the issues related with infinitely repeated multiagent exact belief propagation are discussed. The solution to the issues of multiagent and multithread deadlocks is provided in Section 5. The detailed infinite multiagent belief propagation schemes are presented in Section 6. A running example is provided in Section 7. The performance of the approach under the adverse environments and its possible improvements are discussed in Section 8.
conclusion is made in Section 9.

2 Related Work

In this section, we use “iterative” or “iterated” in italic type to represent the alternative implementation of a recursive process, in contrast to the repeated belief updating at each node or agent.

Pearl’s polytree belief updating algorithm (Pearl, 1988) can be implemented as an iterative process, which presents exact marginals for polytree structured Bayesian networks (BNs) in a number of iterations equal to the diameter of the polytree. The iterated polytree algorithm, when applied to the distributed problem domains, has the similar problem as we discussed with the iterated LJF or DLJF based belief propagation algorithms in Section 1: it is vulnerable to any errors in the belief propagation. The iterated algorithm has been applied to multiply connected BNs to perform approximate inference, which is known as loopy belief propagation (LBP) or iterative belief propagation (IBP) (Murphy, Weiss, & Jordan, 1999; Ihler, Fisher-III, & Willsky, 2005). LBP (IBP) was extended to generalized belief propagation (GBP) (Yedidia, Freeman, & Weiss, 2000) by clustering some of the nodes in BNs into super nodes and applying message passing between the super nodes instead of between the original singleton nodes. GBP improves the approximation of LBP (IBP) whenever it converges. It is possible that GBP converges on the problems that LBP does not. Here, by “converge” we mean the belief iteratively updated reaches a stable equilibrium (which is not guaranteed to be correct, though). Nevertheless, for neither LBP (IBP) nor GBP, there are any general criteria found to predict their convergence or to evaluate the quality of their approximation without computing exact posterior beliefs (Bidyuk & Dechter, 2004). Inspired by GBP, the junction tree inference algorithm of Hugin or Lauritzen-Spiegelhalter architecture (Lauritzen & Spiegelhalter, 1988) was iteratively applied to join-graphs rather than join (junction)-trees to perform approximate inference in general BNs (Dechter, Kask, & Mateescu, 2002), which is called iterative join-graph propagation. Iterative join-graph propagation is a special class of GBP. None of these iterated methods is guaranteed to converge on general BNs. They all are approximate belief updating schemes applied to centralized (single agent) BNs.

Iterative proportional fitting (IPF) iteratively modifies a probability distribution to meet a set of probability constraints while maintaining a minimum KL distance to the original distribution. Our work is different with IPF in that (1) we do not have extra probability constraints that need to be eventually satisfied by iterative operations (we perform belief propagation); (2) not like IPF, belief propagation by our method would not produce the probability distributions that may be inconsistent with the underlying graphical structures; (3) our work tries to address a concurrent message exchanging protocol problem between two or among multiple parties while IPF addresses a problem regarding how extra probability constraints would be fitted by one party; and (4) when applied to multiple parties with the messages exchanged as constraints, the iterative belief updating based on IPF at all agents may not work collectively to reach the correct beliefs.

In (Wong & Butz, 1998), probabilistic asynchronous reasoning in distributed multiagent environments is proposed. This work is different with the work by Xiang et. al. (Xiang, 2002) in that it allows agents share all their knowledge bases. That is, there is no concept of agents’ privacy in the work by Wong and Butz. In the framework proposed by Xiang et. al., agents can only share some limited information with each other and this forms the constraint of many operations in MSBNs, e.g. triangulation (Xiang, 2001), interface verification (Xiang & Chen, 2004), and belief updating (Xiang, 2000). So, the problem addressed by Wong and Butz (Wong & Butz, 1998) can be considered different than the problem addressed by Xiang et. al. (Xiang, 2002). Our work would address the same
problem as the work by Xiang et al. That is, we would respect agents’ privacy. The work (Butz & Wong, 1999) discusses the recovery from agent failures within the framework of (Wong & Butz, 1998). In particular, the work focuses on the concurrency control protocols to address agents’ belief inconsistency problem produced from agents’ failures from time to time, where failures are limited to the downs of agents and agents need to be up eventually. Message errors cannot be accommodated and corrected. Our work discusses how to make agents perform belief updating exactly, recover from wrong beliefs, and adapt to the adverse environments, where deadlock prevention is also addressed in the meantime.

3 Overview of MSBNs

A BN is a triplet $(V, G, P)$, where $V$ is a set of domain variables, $G$ is a directed acyclic graph (DAG) whose nodes are labeled by elements of $V$, and $P$ is a joint probability distribution (JPD) over $V$. $G$ qualitatively encodes conditional independencies in $P$. In an MSBN, a set of $n > 1$ agents $A_0, A_1, ..., A_{n-1}$ populates a total universe $V$ of variables. Each $A_i$ has knowledge over a subdomain $V_i \subset V$ encoded as a Bayesian subnet $(V_i, G_i, P_i)$. The collection $\{G_0, G_1, ..., G_{n-1}\}$ of local DAGs encodes agents’ knowledge of domain dependencies. Local DAGs should overlap and agents exchange information via the shared variables (called interface). Definition 1 gives the definition of hypertree, which organizes Bayesian subnets and agents through the shared variables. In the following discussion, we use a pair $(V, E)$ to denote a graph $G$, where $V$ denotes the set of nodes (vertices) in $G$, and $E$ the set of edges (links). Edges or links could be directed or undirected.

Definition 1 (Xiang, 2002) Let $G = (V, E)$ be a connected graph sectioned into connected subgraphs $\{G_i = (V_i, E_i)\}$. Let these subgraphs be organized into a connected tree $\Psi$ where each node, called a hypernode, is labeled by $G_i$ and each link between $G_i$ and $G_j$, called a hyperlink, is labeled by the interface $V_i \cap V_j$ such that for each pair of nodes $G_i$ and $G_m$, $V_i \cap V_m$ is contained in each subgraph on the path between $G_i$ and $G_m$. The tree $\Psi$ is called a hypertree over $G$.

In a hypertree, each hyperlink can serve as an information exchange channel between agents connected and is referred to as an agent interface. From Definition 1, a hypertree has the property of a junction tree regarding the distribution of the shared variables among hypernodes. If the underlying structure of a hypertree (i.e., $\tilde{G}$) is undirected, the property guarantees that each hyperlink separates the two branches it connects. That is, given the belief on a hyperlink, the beliefs on the two incident branches are guaranteed to be independent of each other. However, if $G$ is directed, this property alone does not guarantee coherent message passing through hyperlinks since a hyperlink does not necessarily separate the two branches it connects. Here, we consider each hyperlink and its two incident branches as three disjoint subsets of nodes in $G$ and discuss the separation property of the hyperlinks based on $G$. A hypertree is undirected, but the separation property of hyperlinks have to be discussed based on its underlying directed structure (i.e., $\tilde{G}$). Note here we do not restrict $G$ to be a DAG so that the hypertree concept can be defined on a wide varieties of graphs. To ensure a hyperlink separates the two hypertree branches connected, the hyperlink has to be a $d$-sepset, as defined in Definition 2.

Definition 2 (Xiang, 2002) Let $G$ be a directed graph such that a hypertree over $G$ exists. A node $x$ contained in more than one subgraph with its parents $\pi(x)$ in $G$ is a $d$-sepnode if there exists a subgraph that contains $\pi(x)$. An interface $I$ is a $d$-sepset if every $x \in I$ is a $d$-sepnode.
The concept \textit{d-sepset} defined in Definition 2 is used to depict the property a hypertree on a DAG needs to satisfy to realize the separation property of hyperlinks. If a node in a hyperlink is not a \textit{d-sepnode}, the two incident branches of the hyperlink would not be separated since the node has at least a different parent at each branch. That is, at least its parents are not separated. For example, as shown in Figure 2, two subnets $A$ and $B$ share a node $a$ which is not a d-sepnode since its parents $b$ and $c$ do not exist in a same subnet. Then the hyperlink that contains $a$ would not separate $A$ and $B$. For the similar reason as in Definition 1, we do not define \textit{d-sepset} only on DAGs, though in this paper, we only use it on DAGs. On DAGs, a \textit{d-sepset} hyperlink separates its two branches.

![Figure 2: A hyperlink with non-d-sepnode does not separate the two incident branches.](image)

The overall structure of an MSBN is a \textit{hypertree MSDAG}. In Definition 3, we define a hypertree on a DAG (sectioned into multiple connected parts — multiple sub-DAGs) where all hyperlinks are d-sepsets. A hypertree MSDAG can be considered a junction tree over the multiple sub-DAGs, which has the equivalent separation property of a junction tree.

**Definition 3** (Xiang, 2002) A hypertree MSDAG $G = \bigcup_{i} G_i$, where each $G_i = (V_i, E_i)$ is a DAG, is a connected DAG such that there exists a hypertree over $G$ and each hyperlink is a d-sepset.

Based on the hypertree MSDAG, an MSBN can be defined as in Definition 4, where a \textit{potential} is a probability distribution without normalization.

**Definition 4** (Xiang, 2002) An MSBN $M$ is a triplet $(V, G, P)$. $V = \bigcup_{i} V_i$ is the total universe where each $V_i$ is a subset of variables, called a subdomain. $G = \bigcup_{i} G_i$ is a hypertree MSDAG where nodes of each subgraph $G_i$ are labeled by elements of $V_i$. Let $x$ be a variable and $\pi(x)$ be all parents of $x$ in $G$. For each $x$, exactly one of its occurrences (a $G_i$ containing $\{x\} \cup \pi(x)$) is assigned $P(x|\pi(x))$, and each occurrence in other subgraphs is assigned a unit constant potential. $P = \prod_{i} P_i$ is the JPD where each $P_i$ is the product of the potentials associated with nodes in $G_i$. Each triplet $S_i = (V_i, G_i, P_i)$ is called a subnet of $M$. Two subnets $S_i$ and $S_j$ are said to be adjacent if $G_i$ and $G_j$ are adjacent on the hypertree.

For a shared variable $x$, only one of its occurrences that contains all its parents $\pi(x)$ is assigned the true belief $P(x|\pi(x))$ on it. All other occurrences are assigned a uniform belief. However, the inconsistent beliefs assigned to different occurrences of a shared variable $x$ will be made consistent in the initial message passing when an MSBN is applied. In an MSBN, each agent holds its partial perspective of a large problem domain, and has access to a local evidence source (sensors). Global evidence can be obtained by communicating with other agents. Agents update their beliefs with local and global evidence, and then answer queries or take actions based on the updated beliefs. Figure 3 illustrates the DAGs of a trivial MSBN in (a) and their hypertree organization in (b). In (a), each dotted box represents a Bayesian subnet, and in (b) each circle denotes a hypernode, and each rectangular box with rounded corner represents a hyperlink.
In an MSBN, only the nodes in agent interfaces are public to the corresponding agents. All other nodes are private and known to the respective agent only. As mentioned above, this forms the constraint of many operations in an MSBN, e.g. triangulation (Xiang, 2001), interface verification (Xiang & Chen, 2004), and belief updating (Xiang, 2000).

4 A Baseline Algorithm

In this section, we present a baseline algorithm and discuss issues that need to be solved for the baseline algorithm to perform fault-tolerant multiagent exact belief updating. In the rest of the paper, we present our method based on an arbitrary agent $A_i$ populating on Bayesian subnet $(V_i, G_i, P_i)$ of an MSBN $M = (V, G, P)$ of $n$ subnets. Denote the adjacent agents of agent $A_i$ ($0 \leq i \leq n - 1$) on the hypertree by $A_j^i$ ($0 \leq j \leq m_i - 1$). Denote the interface $V_i \cap V_j$ between $A_i$ and $A_j^i$ by $I_{ij}$.

Algorithm 4 (FTCoBeliefUpdate_PRI) Each arbitrary agent $A_i$ performs the following operations to continuously update and propagate its belief:

1 while (true)
2 absorb local evidence if any;
3 for each adjacent agent $A_j^i$, do
4 send $A_j^i$ its belief on the interface $I_{ij}$;
5 for each adjacent agent $A_j^i$, do
6 receive the belief of $A_j^i$ on the interface $I_{ij}$;
7 update its belief based on the message received from $A_j^i$;

Algorithm 4 (FTCoBeliefUpdate_PRI) is the baseline algorithm. It roughly describes the basic operations each agent needs to perform. In this algorithm, each agent continuously exchanges messages with its neighbors and updates its own belief based on messages received from its neighbors. Message exchanging among agents can be made asynchronously without necessary loss of packets due to the existence of buffers assigned to data communication protocols (e.g. UDP and TCP) $^1$.

Since each agent keeps updating its own belief and the messages sent to its neighbors, multiagent belief propagation could be potentially made immune to temporary errors in belief updating and

$^1$The size of buffer assigned to a protocol can be set by a privileged user, which has an operating system dependent default value. Even enough buffer is provided, packet loss could occur in delivery with UDP since it is connectionless without error detection. Nevertheless, our method is fault-tolerant.
communication (it turns out not all existing BN belief updating methods can be extended to support fault-tolerant MSBN belief propagation, which we will discuss with more details in sections below). Permanently disconnected communication channels would not prevent properly connected agents from working together. Figure 4 illustrates the basic idea of the algorithm, where each circle denotes a hypernode and each link a hyperlink. There are totally five hypernodes on the hypertree and each hypernode is populated by an agent. Beside each hypernode, each dark headed arrow denotes the messages repeatedly sent out from the hypernode to one of its neighbors and each white headed arrow denotes the messages continuously received by the hypernode from one of its neighbors. Therefore, each dotted box represents an agent and all activities associated with it. Intuitively, since an agent keeps sending and receiving messages and updating its belief based on messages received, any temporary problems on the hypertree would not crush the belief propagation if we can ensure correct messages will eventually reach each agent and agents can update their beliefs based on such messages exactly (no matter what messages it has received previously). In that case, we say the iterated belief updating is exact, fault-tolerant and error-corrective for temporary problems. If there exist any permanent disconnections on the hypertree, all connected portions can continue their belief updating. Although each connected portion cannot get messages from other portions, belief propagation within each portion would be exact, fault-tolerant and error-corrective.

Figure 4: A hypertree of five hypernodes: each dotted box denotes a hypernode and all activities associated with it. The dark headed arrows denote messages repeatedly sent out and the white headed arrows denote messages continuously received.

Next, we discuss how to ensure exact, fault-tolerant and error-corrective belief updating can be done based on the propagation scheme. There are two problems we need to address. Firstly, multi-agent deadlock could happen when some agents are dead or some communication channels become broken since an agent needs to receive one message from each of its neighbors (lines 5 and 6) before sending another round of messages to its neighbors (lines 3 and 4). As shown in Figure 5 is a hypertree with four hypernodes. When channel between A and B is disconnected, B would wait for the message from A before sending out any messages. Without receiving messages from B, A, C and D will stop sending out any messages. The entire system would enter a state of deadlock. Deadlock may take place in each connected portion when the system is permanently disconnected. Note the message sending (lines 3 and 4) has to precede the message receiving (lines 5, 6 and 7) since otherwise deadlock would be immediate. Secondly, we need to investigate what belief propagation architectures can be properly extended to support (possible infinitely) repeated exact belief updating. We will show not all existing MSBN belief updating schemes could be extended to support the iterated exact belief updating.
Multiagent and Multithread Deadlock Prevention

FTCoBeliefUpdate_PRI is a sequential algorithm, where an agent uses a single thread to sequentially perform all operations specified for it. The risk of deadlock could be reduced by designating the total number of messages an agent should receive at each receiving for loop (lines 5, 6 and 7), instead of requiring it to receive one message from each of its neighbors in a loop. As shown in Figure 5, if agent B only needs to receive one message before sending out another round of messages, the channel disconnection between A and B would not result in deadlock since C and D would continue to receive messages and send B messages. Actually even one more communication channel incident to B is broken, the system can still properly move forward. This is presented as another algorithm – Algorithm 5 (FTCoBeliefUpdate_SEQ), where parameter M is used to control the total number of messages an agent needs to receive in a for loop. The smaller the M is, the less possibly the deadlock could occur. When M = 1 is set for all agents, the system is safest from deadlocks since only if there is one hyperlink (communication channel) is alive, an agent can move forward. Note when M = 0, no cooperation exists among agents, where an agent sends messages but does not receive any messages. The messages received in one while loop cycle by agent A_i could come from different while loop cycles of a same neighboring agent A_j.

Algorithm 5 (FTCoBeliefUpdate_SEQ) Denote the number of messages an agent should receive in one while loop cycle by M. Each arbitrary agent A_i performs the following operations to continuously update and propagate its belief:

1. while (true)
2. absorb local evidence if any;
3. for each adjacent agent A_j, do
4. send A_j its belief on the interface I_{ij};
5. for j = 1, ..., M, do
6. receive a message and update its belief based on the message;

The advantage of the sequential implementation of belief updating at an agent is that we only have one thread at one agent. When multiple threads are used at an agent, complicated deadlocks could be resulted. Next, we discuss the concurrent implementation of belief updating at agents.

FTCoBeliefUpdate_PRI can be converted to a concurrent algorithm as shown in Algorithm 6 (FTCoBeliefUpdate_CON). Note we may use Java idioms in the description of our algorithms when necessary. In particular, we use “synchronized” statements from Java to indicate the set of statements will only allow one thread to operate at one time to prevent thread interference and memory consistency problems (Tanenbaum, 2001). In this variant, at each agent A_i, two threads are created for each neighbor A_j (lines 1, 2 and 3), one for keeping sending A_i’s belief to A_j (lines 4 through 7) and the other for uninterruptedly receiving the messages from A_j and updating A_i’s belief B_i based on
the messages received or local evidence observed if any (lines 8 through 11). Note a thread needs
to sleep a random time in each while loop cycle to prevent possible starvation of some other threads.
To prevent thread interference on critical resource $B_i$ and memory consistency errors, critical regions
(lines 5 and 10) are synchronized.

**Algorithm 6 (FTCoBeliefUpdate_CON)** Each arbitrary agent $A_i$ performs the following operations to set up environment and continuously update and propagate its belief $B_i$:

1. for $j = 0, ..., m_i - 1$, do
2. create a thread $T_{is}^j$ for sending messages to neighbor $A_i^j$;
3. create a thread $T_{ip}^j$ for processing messages from neighbor $A_i^j$;

Each $T_{is}^j$ performs the following operations:
4. while (true)
5. synchronized($B_i$){obtain $A_i$’s belief $S_i^j$ on $I_i^j$; }
6. send $S_i^j$ to $A_i^j$;
7. sleep a random time;

Each $T_{ip}^j$ performs the following operations:
8. while (true)
9. receive the belief $R_i^j$ of $A_i^j$ on $I_i^j$;
10. synchronized($B_i$){update $B_i$ based on $R_i^j$ and local evidence if any; }
11. sleep a random time;

Figure 6: An agent has a sending thread and a receiving thread corresponding to each neighbor.

In the concurrent version, no multiagent deadlock will happen because the main threads of agents
are only responsible for local environment settings (lines 1 through 3). The potential deadlock issues
between threads from adjacent agents can be solved through communication protocols, typically using
semaphores (Tanenbaum, 2001), which is out of the scope of the paper, however. Generally, there are
two pairs of threads for each pair of adjacent agents. Each pair of threads is responsible for message
passing in only one direction, as shown in Figure 6, where two adjacent hypernodes 1 and 2 each has
two threads for the other, one for sending messages to and one for receiving messages from the other.
The sending thread (e.g., $T_{is}^1$) and the receiving thread from the other (e.g., $T_{ip}^2$) are considered a pair
for message passing in one direction. The algorithm does not introduce any new issues that could
cause deadlock between each pair of such threads. Will the synchronized multiple threads (e.g., $T_{is}^1$ and
$T_{ip}^2$ in Figure 6) at the same agent cause deadlocks? The answer is no because such threads are
generally independent with each other and there is no such things like hold and wait or circular wait.
In particular, in some programming languages such as Java, the lock will always be released when
an exception happens. Therefore, the problematic threads would not affect other threads on the same
agent when such locks are used. The disconnected communication between a pair of threads would
not interfere with other threads’ work either. That is, the system is robust and is able to self-adapt to
the environment.
6 Infinite Exact Belief Propagation

In this section, we discuss the extension of the belief propagation architectures to support (infinitely) repeated exact belief updating. As we mentioned above, not all existing MSBN belief propagation schemes could be extended to support iterative belief propagation. We study the possible extension for each scheme in this section.

There currently exist four junction tree (JT) based probabilistic belief propagation architectures. They are Lauritzen-Spiegelhalter architecture, Hugin architecture, Shenoy-Shafer architecture, and Lazy architecture. Hugin, Shenoy-Shafer and Lazy architectures have all been extended to LjF or DLjF based inference in MSBNs (Xiang, 2002; Xiang & Jensen, 1999; Xiang et al., 2006). Lauritzen-Spiegelhalter architecture is similar with Hugin architecture, and their difference is mainly in that, in Hugin separators are used to store messages between adjacent cliques, which makes computation more efficient (Lepar & Shenoy, 1999). Hence, Lauritzen-Spiegelhalter can be extended to LjFs similarly as Hugin. Like their originals, these extended architectures differ in their potential representation, storage, message passing and belief updating in each JT of LjFs or DLjFs. When message passing between adjacent agents is needed, belief propagation through linkages (from linkage trees or message JFs) is performed. Message JFs in DLjFs are generally direction-dependent (a different set of JFs is used for message passing in different directions of a hyperlink), which is different with the direction-independent linkage trees (the same linkage tree is used for message passing in both directions) in LjFs. The belief propagation via linkages is the major difference between the extended and the original architectures, which involves linkage belief acquisition, absorption, and propagation. Next, we specifically discuss which architecture extensions can be iterated to support infinite exact belief inference in LjFs.

![Figure 7: A linkage tree for a pair of JTs.](image)

6.1 Iterated Hugin Extension

Hugin architecture has been extended to LjFs, which have a consistent linkage tree for each pair of adjacent agents (Xiang, 2000, 2002). The linkage tree over a d-sepset can be derived from the agent’s local JT as detailed in (Xiang, 2000), which expresses the same graphical separation on the d-sepset as its deriving JT. Note the linkage tree derived by one agent over a d-sepset is equivalent to that derived by the other agent on the same d-sepset: they have identical linkages and separators. A linkage in a linkage tree is a subset of at least one clique of its deriving JT. One of such cliques is designated as the host of the linkage. Figure 7 shows a linkage tree constructed from a pair of JTs, where each dotted box denotes a local JT and between them is the linkage tree. Algorithms 7 and 8 describe how belief propagates via linkage trees in the extended Hugin architecture, where $A_i$ and $A_j$ are two adjacent agents, $A_i$ is associated with local JT $J_i$ and linkage tree $L_i$, and $A_j$ is associated with JT $J_j$ and linkage tree $L_j$. Note $L_i$ and $L_j$ are identical in their structures. That is, messages are passed through linkages one by one. For each linkage, the caller agent asks the callee agent to calculate its
belief on the linkage and send the belief to the caller. The caller then updates its belief on the linkage and its belief on the host of the linkage. Finally, the caller calls UnifyBeliefHugin to propagate the its belief on the host of linkage to all other cliques. Here, UpdateBeliefHugin is the Hugin version of UpdateBelief called in Algorithms 2 and 3, and UnifyBeliefHugin called by UpdateBeliefHugin is the Hugin version of UnifyBelief called in Algorithm 2. As we mentioned in Section 1, such message passing and belief updating methods are too fragile to the practical environments. If any wrong messages are passed and absorbed by any agent at any time point, the agents’ beliefs would become inconsistent. Any communication problems would make the whole system freeze.

**Algorithm 7 (UpdateBeliefHugin)** (Xiang, 2002) When UpdateBeliefHugin is called on $A_i$ relative to $A_j$, the following occurs:

1. for each linkage $Q_j$ with host $C_j$ in $L_j$, $A_j$ assigns $B_{Q_j}(Q_j) = \sum_{C_j \setminus Q_j} B_{C_j}(C_j)$;
2. for each linkage $Q_i$ with host $C_i$ in $L_i$, $A_i$ calls AbsorbViaLinkageHugin for $C_i$ to absorb via $Q_i$;
3. $A_i$ performs UnifyBeliefHugin at $J_i$.

**Algorithm 8 (AbsorbViaLinkageHugin)** (Xiang, 2002) Let $Q_j$ be the linkage in $L_j$ that corresponds to $Q_i$ in $L_i$. When AbsorbViaLinkageHugin is called on $A_i$ for $C_i$ to absorb through $Q_j$, the following occurs:

1. $A_i$ requests transmission of $B_{Q_j}(Q_j)$ from $A_j$;
2. $A_i$ updates its host potential $B_{C_i}(C_i) = B_{C_i}(C_i) * B_{Q_j}(Q_j)/B_{Q_i}(Q_i)$;
3. $A_i$ updates its linkage potential $B_{Q_i}(Q_i) = B_{Q_j}(Q_j)$.

**Algorithm 9 (iTransmitBeliefHugin)** Denote the belief of $A_i$ by $B_i$. When iTransmitBeliefHugin is called on $A_i$ relative to $A_j$, $A_i$ does the following:

1. synchronized($B_i$) for each linkage $Q_i$ with host $C_i$, $A_i$ assigns $B_{Q_i}(Q_i) = \sum_{C_i \setminus Q_i} B_{C_i}(C_i)$;
2. send $L_i$ to $A_j$.

**Algorithm 10 (iUpdateBeliefHugin)** When iUpdateBeliefHugin is called on $A_i$ relative to $A_j$, $A_i$ does the following:

1. synchronized ($B_i$ & linkage potentials from $A_j$) for each linkage $Q_i$ with host $C_i$ in $L_i$, do
2. update its host potential $B_{C_i}(C_i) = B_{C_i}(C_i) * B_{Q_j}(Q_j)/B_{Q_i}(Q_i)$;
3. update its linkage potential $B_{Q_i}(Q_i) = B_{Q_j}(Q_j)$;
4. perform UnifyBeliefHugin at $J_i$.

To make Hugin extension robust, we propose to modify it by allowing all agents in the domain to repeatedly send and receive messages to and from their neighbors. Hence, the collection process and the distribution process that are used to facilitate the traversal of the hypertree and message passing in the extension are not needed any more, and its main algorithm can be replaced by Algorithm 6. Message transmission and belief updating performed by Algorithms 7 and 8 can be achieved by Algorithms 9 and 10 collectively. By iTransmitBeliefHugin, an agent repeatedly sends messages to its neighbors, and by iUpdateBeliefHugin, an agent repeatedly updates its belief based on run-time messages received. After replacing lines 5 and 6 with iTransmitBeliefHugin and lines 9 and 10 with iUpdateBeliefHugin, FTCoBeliefUpdate becomes an iterated Hugin extension on JLFs.
However, Hugin architecture or its extension on LJFs are originally designed for partially ordered belief propagations (i.e. there is a partial order regarding belief updating on cliques or hypernodes), where old potentials on separators or linkage trees need to be kept for belief updating (In line 2 of Algorithm 8, $B_{Q_i}(Q_i)$ is the old potential of $Q_i$ and $B_{Q_j}(Q_j)$ corresponds to the new potential of $Q_i$). Then a question is how the old potentials on linkage trees can be maintained and used in iterated Hugin extension such that belief updating can be properly performed (line 2 of Algorithm 10). In partially ordered belief updating, only one copy of old potential needs to be maintained for a linkage tree. However, in the infinitely iterated asynchronous belief updating as specified by the set of Algorithms 6, 9 and 10, two adjacent agents could simultaneously send messages to each other. It is impossible for both agents to maintain a consistent copy of the old linkage tree potentials. Will maintaining a copy of old linkage tree potential for each direction help? The answer is no because there is no guarantee on the convergence of agents’ beliefs updated such, and if they converge, they may not converge to the correct posterior probabilities. We illustrate the problems faced by the proposal by an example below.

Figure 8: A two clique JT.

### Table 1: $P(a, b)$ and $P(b, c)$ are consistent on separator $b$.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>$P(a, b)$</th>
<th>c</th>
<th>$P(b, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.02</td>
<td>0</td>
<td>0.35</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.18</td>
<td>0</td>
<td>0.15</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.48</td>
<td>1</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.32</td>
<td>1</td>
<td>0.40</td>
</tr>
</tbody>
</table>

### Table 2: $P(a, b)$ and $P(b, c)$ after absorbing messages from each other.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>$P(a, b)$</th>
<th>c</th>
<th>$P(b, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.28</td>
<td>0</td>
<td>0.28</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.72</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.72</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For simplicity, we illustrate the problems on a two clique JT. Since a LJF is a JT in its nature, the conclusion obtained here also applies to LJFs. The JT as shown in Figure 8 has two cliques \{a,b\} and \{b,c\}. Two unit constant potentials are assigned to its separator, one in each direction. We assume that the two cliques have reached belief consistency by an initial belief propagation. Their beliefs are as shown in Table 1. Assume evidence $a = 0$ is then observed in clique $ab$ and evidence $c = 0$ in clique $\{b,c\}$. The two cliques send new messages to each other, and their beliefs are updated as shown in Table 2 (after normalization). At this time, the two messages kept for the separator $b$, one for each direction, are $P_{\{a,b\}}(b) = \{0.02, 0.18\}$ from clique \{a,b\} to \{b,c\} and $P_{\{b,c\}}(b) = \{0.35, 0.1\}$ in the other direction. If such belief exchanging and updating continues infinitely, the beliefs on the
two cliques will converge to $P(b) = \{0, 1\}$ \(^2\), which is not the correct result \(^3\).

For the similar reason, we may not be able to properly iterate Lauritzen-Spiegelhalter extension for repeated belief inference with LJFs.

### 6.2 Iterated Shenoy-Shafer Extension

Shenoy-Shafer architecture on join trees (Lepar & Shenoy, 1999) has been extended to both LJFs and DLJFs (Xiang & Jensen, 1999). It turns out extended Shenoy-Shafer architecture can be further extended to support iterated exact belief propagation. Intuitively, extended shenoy-shafer architecture supports iterative multiagent exact belief propagation because that no matter on each local JT level or on the LJF or DLJF level, the input potentials are never altered during belief propagation. The belief of a clique is obtained by multiplying its input potentials with messages received from each of its neighboring cliques. The message sent from a clique $C_i$ to a neighboring clique $C_j$ is obtained by multiplying the input potentials of $C_i$ with the messages received from all other neighbors marginalized to the separator $C_i \cap C_j$. In all the process, a copy of input potentials is always maintained. Since the belief of a clique and all messages sent from the clique are dynamically formed by its input potentials and the messages received on the fly, we can update the belief of the clique and the messages it needs to send out at any time. We only need to ensure that all messages will become right eventually. Similarly, on the LJF or DLJF level, an agent’s belief is represented by a local JT or a local JF, which always maintains a copy of input potentials at each clique. The belief of an agent or the messages sent from the agent change only when the agent receives different messages. Since in iterative version, each agent iteratively updates the messages it sends out, these messages may not be always correct.

\[ \text{Figure 9: Iterative messages may not always be correct.} \]

For example, as shown in Figure 9, before the messages from agents $A$ and $E$ reach agent $B$ and are used to update the message sent from $B$ to $C$, any messages sent from $B$ to $C$ are not guaranteed to be correct (before the initial message from a neighboring agent $Y$ arrives, an agent $X$ assumes a uniform potential message from $Y$ available in updating its messages to all other agents). Similar issues exist at agent $C$. In particular, after $B$ or $C$ receives incorrect message from each other, any messages sent from them to all other neighboring agents would not be correct. We need and only need to ensure all messages would converge to the right ones (as obtained by the non-iterative version).

A simple analysis on the scheme indicates that all messages continuously exchanged among agents would eventually converge to the right ones, which is summarized as Proposition 1. Note in this proposition, we assume messages can be properly exchanged among agents, and based on the assump-

---

\(^2\)Without normalization, $P(b = 0)$ and $P(b = 1)$ each forms a number series as two cliques iteratively exchange and update their beliefs. Assume at the 1st iteration, $P_1(b = 0) = 0.28 = \frac{7}{25} \times \frac{10}{9}$, and $P_1(b = 1) = 0.72 = \frac{7}{25} \times \frac{9}{10}$. Then, at the $n'th$ iteration, $P_n(b = 0) = \left(\frac{7}{9}\right)^{(n-1)} \mod 2 \frac{7}{9} \times \frac{10}{9}$, and $P_n(b = 1) = \left(\frac{2}{9}\right)^{(n-1)} \mod 2 \frac{2}{9} \times \frac{9}{10} \frac{10}{9}$. Therefore, $\lim_{n \to \infty} P_n(b=0)=0$, and with normalization, $\lim_{n \to \infty} P_n(b = 0) = 0$, and $\lim_{n \to \infty} P_n(b = 1) = 1$.

\(^3\)The correct result is $P(b) = \{0.28, 0.72\}$. 

---

15
tion, we discuss the property of the iterated Shenoy-Shafer message passing. Messages exchanging details would be discussed subsequently.

![Figure 10: Correct messages propagate and overwrite old messages on a hypertree: (a) the leaf hypernodes send messages to the non-leaf hypernodes; (b) a smaller hypertree $H'$ is obtained by removing the leaf hypernodes in hypertree $H$ in (a); (c) the hypertree would eventually become the form in (c); (d) the correct messages are backward propagated toward initial hypernodes. Each circle denotes a hypernode and the irregular boxes denote the part of the hypertree exclusive of the leaf hypernodes. Ellipses denote the omitted branches of the hypertree. Each arrow indicates the direction of the message concerned.](image)

**Proposition 1** An MSBN is populated by a set of agents with one agent on a hypernode. Assume each agent maintains a copy of input potentials, and its message sent to a neighboring agent is formed by multiplying its local input potentials and the incoming messages from all other neighboring agents marginalized to the corresponding hyperlink. The incoming message from an neighboring agent is initialized to be a uniform potential before any messages from the neighboring agent are received. If agents continuously send messages formed such to their neighboring agents, the messages would converge to the ones as obtained by the respective extended Shenoy-Shafer architecture on MSBNs (Xiang & Jensen, 1999).

Proof:

As shown in Figure 10 (a), a hypertree always has some hypernodes with only one incident hyperlink, which we call the leaf hypernodes (e.g. hypernodes $A$, $B$, $C$, and $D$). The messages iteratively sent out by the leaf hypernodes would always be correct since such messages are formed completely based on the input potentials of and would be irrelevant with any messages received by the respective leaf hypernodes. We assume such messages can be properly received by their receivers, and these receivers would become a set of the leaf hypernodes if we remove the initial set of the leaf hypernodes from $H$, as shown in (b). The messages sent out by the new set of the leaf hypernodes (e.g., $A'$, $B'$, $C'$, and $D'$ in (b)) toward the irregular box would always be correct since such messages are formed by the input potentials of the respective new leaf hypernodes and the correct messages received from the previous leaf hypernodes, and would be irrelevant with the messages received from the irregular box. We perform the onion peeling analysis until we get a hypertree as shown in (c). In that case, all messages obtained by $L$ from its neighboring agents would be correct, and once $L$ receives such messages, all messages sent out by $L$ to any of its neighboring agents would be correct. This would further make all messages sent away from $L$ by the neighboring agents of $L$ correct, as shown in (d). Until the backward propagation process reaches the initial leaf hypernodes of hypertree $H$, all messages exchanged among agents would be correct ones as obtained by the respective extended Shenoy-Shafer architecture on MSBNs (Xiang & Jensen, 1999).

Intuitively, by the iterated Shenoy-Shafer architecture, all the leaf hypernodes keep pumping correct messages into the hypertree while such messages would never be impaired by any possibly wrong
messages received. When all forward messages from the leaf hypernodes to the core of the hypertree become right, the backward messages from the core to the leaf hypernodes would start to become right since from that moment, the core also keeps pumping right messages toward the leaf hypernodes.

Next, we need to provide specific message exchanging details to further validate the method. However, we would not provide such details separately for the iterated Shenoy-Shafer extensions. Instead, we would discuss such details jointly with the iterated Lazy extensions. This is because lazy propagation performs belief updating and message passing based on the scheme of Shenoy-Shafer propagation on JT (Madsen, 2006), and this is true for both the extended Lazy propagation and the extended Shenoy-Shafer propagation on LJF or DLJF (Xiang & Jensen, 1999; Xiang et al., 2006). Particularly, the probability distribution tables assigned to cliques in the Lazy architecture and its extensions are multiplied (combined) only when necessary, which is more complex than always maintaining potentials as products as in the Shenoy-Shafer architecture and its extensions.

6.3 Iterated Lazy Extensions

In this section, we discuss the convergence problems of the iterated Lazy extensions. Before discussing them, we first take a look at the Lazy extensions on MSBNs.

6.3.1 Lazy Extensions on MSBNs

In the Lazy architecture, each clique in a JT is assigned a set of probability tables, and the product of these tables is not calculated. Lazy propagation has been extended to both LJFs and DLJFs, which consists of a round of inward propagation and a round of outward propagation along the hypertree. In the extended Lazy propagations (Xiang et al., 2006), each agent performs lazy propagation on the local JT or local junction forest (JF). Messages are propagated among agents via linkages as specified in Algorithms 11-14, where “adjacency” is defined such that two cliques are adjacent if they are directly connected in a JT or are hosts of the same linkage. For example, as shown in Figure 7, clique \{a,b,e\} is not only adjacent with cliques \{e,f\} and \{b,c,d\}, but is also adjacent with clique \{a,b,h\} in a different subnet since they share the same linkage \{a,b\}. Note for a linkage, there is only one host in a subnet. That is, for a linkage, there is only one host in a JT in LJFs or a message junction forest (JF) in DLJFs. Message exchanging between a pair of adjacent agents is eventually through such host pairs. The “adjacency” definition makes message passing over linkages by SendPotentialLazy possible since both a clique C and one of its adjacent cliques can be the hosts of the same linkage for a pair of hypernodes. That is, SendPotentialLazy can be used to propagate messages either within or beyond local JT. In Algorithm 14, the marginalization out of a set \(U \setminus V\) of variables from a set \(B(U)\) of potentials over a set \(U\) of variables is done by marginalizing out these variables one by one (Madsen & Jensen, 1999) and is denoted by

\[
B(V) = \sum_{U \setminus V} B(U).
\]

Note in extended Shenoy-Shafer propagation, \(B(U)\) and \(B(V)\) can be considered the potential products, which would not change our following discussions. In the following discussion, we also assume a clique in a JT is assigned a buffer, called in-buffer, on each separator incident to it to receive the potentials on the separator from the corresponding adjacent clique.

Note Algorithms 11-13 are the extended Lazy version on MSBNs of Algorithms 1-3. In the extended Lazy version, the messages passed among agents are not absorbed immediately, so there does not exist the corresponding Lazy version for UnifyBelief called in Algorithm 2. However, there
does exist a Lazy version for \textit{UpdateBelief}, which is called \textit{SendPotentialLazy}. \textit{SendPotentialLazy} is used to deal with message exchanging among adjacent agents. It is different from the Hugin version \textit{UpdateBeliefHugin} (Algorithm 7) of \textit{UpdateBelief} in that (1) \textit{SendPotentialLazy} can also be used for message passing in local JTs; (2) the products of the potentials in \textit{SendPotentialLazy} are not calculated; and (3) the messages are not absorbed immediately. The set of algorithms would also work for the \textit{extended} Shenoy-Shafer propagation (Xiang \& Jensen, 1999), except we need to use the products of the potentials, instead of the sets of potentials, to represent beliefs in Algorithm 14.

\textbf{Algorithm 11 (UnifyPotentialLazy)} (Xiang et al., 2006) To perform extended Lazy propagation on a JT, the following occurs:

1. select a clique $S$ arbitrarily;
2. call \textit{CollectPotentialLazy} on $S$;
3. call \textit{DistributePotentialLazy} on $S$;

\textbf{Algorithm 12 (CollectPotentialLazy)} (Xiang et al., 2006) When \textit{CollectPotentialLazy} is called on clique $C$, $C$ does the following:

1. for each adjacent clique $Q$ not connected through a linkage except caller, call \textit{CollectPotentialLazy} on $Q$ and get potentials sent from $Q$;
2. \textit{SendPotentialLazy} relative to caller if it is an adjacent clique.

\textbf{Algorithm 13 (DistributePotentialLazy)} (Xiang et al., 2006) When \textit{DistributePotentialLazy} is called on clique $C$, for each adjacent clique $Q$ not connected through a linkage except caller, $C$ does the following:

1. \textit{SendPotentialLazy} relative to $Q$;
2. call \textit{DistributePotentialLazy} on $Q$.

\textbf{Algorithm 14 (SendPotentialLazy)} (Xiang et al., 2006) Let $C_i$ be a clique in a JT, whose adjacent cliques are denoted by $C_{i1}^1, C_{i2}^1, \ldots, C_{im}^1$, respectively. Let $B_i$ be the set of potentials assigned to $C_i$, and $B_{ji}^1$ be the set of potentials in the respective in-buffer sent from $C_{ij}^1$ ($1 \leq j \leq m$). When \textit{SendPotentialLazy} is called in $C_i$ relative to $C_{ik}^1$, $C_i$ does the following:

1. $B_k = \sum_{C_i \setminus C_k} (B_1 \cup_{j \neq k} B_{1}^j)$;
2. send $B_k$ to $C_{ik}^1$.

Next, we discuss the convergence problems of the two iterated Lazy extensions on MSBNs separately.

\subsection{Iterated Lazy Extension on LJFs}

In Lazy propagation on JTs, only marginalization and multiplication are used in potential manipulation, in the extended Lazy propagation on LJFs, division is needed, however. This is because message passing among agents may be done via multiple linkages (not like in JTs, message passing is done via single-clique separators) and hence duplicated beliefs could be acquired by agents. Potential division is defined as Definition 5.
Definition 5 (Xiang et al., 2006) The Lazy division of a set $A$ of potentials by another set $B$ of potentials denoted $A / _L B$ is equal to $A$ after the following operations: (1) if a potential appears in both $A$ and $B$, remove it from both; (2) for each potential $P \in B$, multiply the set $P$ of potentials in $A$ whose domains overlap with that of $P$, divide the product by $P$, and replace $P$ in $A$ by the result of the division.

Algorithm 15 (SendLinkageMsgLazy) (Xiang et al., 2006) Potentials are passed from JT $J$ to JT $J'$ via the directed linkage tree $L$ as follows:

1. each linkage $Q$ in linkage tree $L$ requests its host in $J$ to SendPotentialLazy relative to $Q$, and denote the set of potentials received by $Q$ by $P_Q$.
2. after $Q$ receives potentials from its parent in $L$, send $P^*_{Q'} = \sum_{Q \backslash Q'} P_Q$ to each child $Q'$ of $Q$.
3. send $P_Q / _L P^*_{Q}$ to the host of $Q$ in $J'$.

Algorithm 16 (FTCoBeliefUpdateLazyLJF_CON) Each arbitrary agent $A_i$ with JT $J_i$ performs the following operations to set up the environment and continuously propagate Lazy beliefs:

1. for $j = 0, \ldots, m_i - 1$, do
2. create a thread $T^j_{is}$ and a copy $J_{ij}$ of $J_i$ for sending messages to neighbor $A^j_{ij}$;
3. create a thread $T^j_{ip}$ for receiving messages from neighbor $A^j_{ij}$;
4. while (true)
5. synchronized(all linkage potentials){
6. perform UnifyPotentialLazy using linkage potentials from all adjacent agents at $J_i$;
7. sleep a random time;

Each $T^j_{is}$ performs the following operations:
8. while (true)
9. synchronized(all linkage potentials except those from $A^j_{ij}$){
10. perform UnifyPotentialLazy using linkage potentials from all adjacent agents except $A^j_{ij}$ at $J_{ij}$;
11. perform SendLinkageMsgLazy using $J_{ij}$ relative to the JT associated with $A^j_{ij}$;
12. sleep a random time;

Each $T^j_{ip}$ performs the following operations:
13. while (true)
14. synchronized(potential sets on $I^j_{ij}$){receive the potential set on each linkage from $A^j_{ij}$;}

Potential division is used to remove possible duplicated potentials in message passing between agents. When two adjacent agents communicate, potentials are sent from one agent with JT $J$ to the other agent with JT $J'$ via the linkage tree. The potentials on linkages are obtained from their hosts in $J$. Before potentials on linkages are calculated, UnifyPotentialLazy needs to be performed to ensure linkage hosts have updated potentials. As a result, the set of potentials sent from linkages with shared variables contains duplicated messages. Algorithm 15 ensures no duplicated information will be passed via linkages, where the linkage tree $L$ from $J$ to $J'$ is directed arbitrarily such that each linkage has at most one parent in $L$. For more details, readers can refer to (Xiang et al., 2006).
To iterate the Lazy extension on LJFs, Algorithm 6 (FTCoBeliefUpdate, CON) needs to be modified to suit the Lazy concept, which is originally presented based on Hugin concept. Algorithm 16 (FTCoBeliefUpdateLazy,LIF, CON) is the updated algorithm for iterated Lazy extension on LJFs, where at each agent, two threads are created for each adjacent agent, one for sending messages to, and the other for receiving messages from, the adjacent agent (lines 1 through 3). For a sending thread, a copy of the JT associated with the host agent is created for computing messages to the neighbor. Note linkage potentials sent from adjacent agents are visible to and shared by all sending threads. Then the main thread continuously propagates the messages received from all neighbors within the subdomain (lines 4 through 7). Each receiving thread is responsible for receiving messages from the corresponding sending thread at the other agent (lines 13 and 14). Line 14 is synchronized to maintain message integrity. Each sending thread is responsible for computing and sending messages to the neighbor (lines 8 through 12). The message preparation and forwarding is done by SendLinkageMsgLazy, which does not need to be synchronized since it works based on a JT copy used exclusively. In the iterated version, all in-buffers of a linkage host in a JT need to be initialized with unit constant potentials since they could be used before any messages are received from the respective adjacent agents. For Algorithm 16, we have Proposition 2. Compared with the proof for Proposition 1, the proof for Proposition 2 focuses on examining the message exchanging among agents and the converging speed of such messages. The idea used to prove message converging in Proposition 2 is also a little different from that used for proving Proposition 1.

**Proposition 2** By the iterated Lazy extension on LJFs specified by Algorithm 16, all agents’ beliefs would converge to the exact ones as obtained by the extended Lazy propagation on LJFs (Xiang et al., 2006).

Proof:

We first show messages can be properly exchanged among agents in the iterated Lazy extension on LJFs. Denote the subdomain of A_i by V_i, and the linkage tree between A_i and A_j by L_{ij}. We consider A_i sends messages to A_j via L_{ij}. By performing UnifyPotentialLazy at line 10, we get potential B'(V_i) = \prod_{C \in L_j} B(C) \prod_{Q \in L_j} B(Q), where B(C) is the set of potentials assigned to clique C in JT J_i, and B(Q) is the set of potentials assigned to linkage Q in linkage trees other than L_{ij}. That is, B'(V_i) is calculated based on the set of potentials at JT J_i of A_i and the message potentials received by A_i from all neighboring agents other than A_j. By performing SendLinkageMsgLazy at line 11, we get potential B(I_i) = \prod_{Q' \in L_j} B(Q') sent from A_i to A_j via linkage tree L_{ij}, where B(Q') is the set of potentials assigned to Q'. Proposition 6 in (Xiang et al., 2006) indicates that B(I_i) = const \sum_{V_i \not\subseteq I_i} B'(V_i), where const stands for a positive constant. That is, by message passing operations specified in Algorithm 16, the messages repeatedly sent over a linkage tree are the sets of marginal potentials over the respective d-sepset.

Next, we show messages do converge to the right ones as obtained by the Lazy extension on LJFs (Xiang et al., 2006). Let the diameter of the hypertree (the number of edges of a longest path in the hypertree) be d, and the maximum time needed for a pair of iterative agents to finish a message exchanging over their d-sepset be 1. Then all linkage tree potentials on the hypertree need at most a time d to converge to the right ones. This is because: (1) The set of linkage tree potentials sent from A_i to A_j would only be updated based on the sets of potentials assigned to hypernodes on the hypertree branch of A_i. For example, as shown in Figure 11, only the sets of potentials assigned to A_1 to A_4 (all agents within the irregular box) contribute to the updation of M_{1,0} (all linkage trees are initialized with sets of unit constant potentials for each direction). (2) If we direct a hypertree from all other hypernodes towards a hypernode G_i, then with one more time unit, the converged linkage
tree potentials would expand forward from the leaf hypernodes towards \( G_i \) by at least one hypernode. For example, as shown in Figure 11, after a time unit, \( M_{4,3} \) and \( M_{2,1} \) should have converged to the right ones since they have nothing to do with the messages received by \( A_4 \) or \( A_2 \). After another time unit, \( M_{3,1} \) should have become right, and at the end of the third time unit, all messages sent to \( A_0 \) from \( A_1 \) should be always right (on the condition that no new evidence would be entered). Therefore, after a time \( d \), all linkage tree potentials on the hypertree should have converged to the right ones. All agents’ beliefs would also converge to the right ones. □

![Figure 11: A partial hypertree, where each circle denotes a hypernode (associated with an agent), and each edge denotes a hyperlink. The irregular box represents one of the hypertree branches that correspond to the hyperlink between \( A_0 \) and \( A_1 \). On a hyperlink, the messages destined to different directions are represented by different copies of the corresponding linkage tree: \( M_{i,j} \) stands for the set of potentials as a message sent from \( A_i \) to \( A_j \). We also use arrows to represent messages for different directions: the arrows with dark head and solid shaft represent the messages that contribute to \( M_{1,0} \), and those with white head and dotted shaft represent the messages that benefit from \( M_{0,1} \).](image)

6.3.3 Iterated Lazy Extension on DLJFs

Like Lazy propagation on JTs, only marginalization and multiplication operations are needed for Lazy inference on DLJFs (Xiang & Jensen, 1999; Xiang et al., 2006). There is no need for division operations in Lazy propagation with DLJFs. This is because with DLJFs, for message passing to a neighbor, a triangulated Bayesian subnet is organized into a set of JTs, called a junction forest (JF). In each JT of the JF, there is one and only one clique used for message extraction and passing and initial belief assignment for DLJFs is done as for LJFs. Therefore, for message passed via a JF, there is no problem of duplicated messages. In this section, we discuss if the Lazy extension on DLJFs can be made iterated infinitely.

We first introduce how DLJFs are obtained and manipulated. Algorithms 17-19 describes how a hypertree MSDAG is triangulated, where the caller \( A^c_i \) is either the system coordinator or an adjacent agent of \( A_i \). The other adjacent agents of \( A_i \) are denoted by \( A^1_i, \ldots, A^n_i \). Some properties of the triangulation are: (1) the d-sepsets are not necessarily triangulated (eliminable) in both directions; (2) the triangulation result is not dependent of the starting agent specified in Algorithm 17; (3) the triangulation is analogous to Lazy propagation: without designating a starting agent, an agent can start to triangulate its subdomain relative to a d-sepset once it receives fill-in information on all other d-sepsets produced by its neighbors, and the triangulation is done based on such fill-ins (so, only the leaf hypernodes can be triangulated in the beginning); and (4) there is a separate triangulation relative to each d-sepset.
Algorithm 17 (CommunicateFillin) (Xiang et al., 2006) The system coordinator does the following:
1. select an agent $A_i$ arbitrarily;
2. call CollectFillin in $A_i$;
3. call DistributeFillin in $A_i$ with empty fill-ins;

Algorithm 18 (CollectFillin) (Xiang et al., 2006) Denote the local moral graph of an agent $A_k$ by $G_k = (V_k, E_k)$ ($0 \leq k < n$). When $A_i$ is called by $A_i^c$ to perform CollectFillin, it does the following:
1. initialize accumulator $F = \emptyset$;
2. for $j = 1$ to $m$, do
3. call CollectFillin in $A_i^c$ and receive fill-ins $F_i^j$ on d-sepset $V_i^j \cap V_i^c$;
4. update $F = F \cup F_i^j$;
5. if $A_i^c$ is an adjacent agent, do
6. eliminate $V_i^c \setminus V_i^j$ from $G_i = (V_i, E_i \cup F)$ in an order $O_{i\rightarrow c}$ and add fill-ins to $F$;
7. denote $G_{i\rightarrow c} = (V_i, E_i \cup F)$ and send $F_{\downarrow V_i^c \cap V_i^c} \cap V_i^j$ to $A_i^c$;

Algorithm 19 (DistributeFillin) (Xiang et al., 2006) Denote the local graph produced by CollectFillin at $A_i^j$ by $G_i^j = (V_i^j, E_i^j)$, and the set of fill-ins received by $A_i$ from $A_i^j$ by $F_i^j$. When $A_i$ is called by $A_i^c$ to perform DistributeFillin with fill-ins $F_i^c$, it does the following:
1. for $j = 1$ to $m$, do
2. denote $F = F_i^c \cup_{k=1,k\neq j}^{m} F_i^k$;
3. eliminate $V_i^c \setminus V_i^j$ from $(V_i, E_i \cup F)$ in an order $O_{i\leftarrow j}$ and add fill-ins to $F$;
4. denote $G_{i\leftarrow j} = (V_i, E_i \cup F)$;
5. call DistributeFillin in $A_i^j$ with fill-ins $F_{\downarrow V_i^c \cup V_i^j}$;

Algorithm 20 (BuildMessageJF) (Xiang et al., 2006) Let $A_i$ and $A_j$ be two adjacent agents over subdomains $V_i$ and $V_j$ respectively. Let $G_{i\rightarrow j}$ be the graph at $A_i$ such that $V_i \setminus V_j$ is eliminable. When $A_i$ is called to BuildMessageJF $W_{i\rightarrow j}$ relative to $A_j$, $A_i$ does the following:
1. identify set $L_{i\rightarrow j}$ of cliques in subgraph of $G_{i\rightarrow j}$ spanned by $V_i \cap V_j$;
2. if $L_{i\rightarrow j}$ is a singleton, create a JF $W_{i\rightarrow j}$ with single JT from cliques of $G_{i\rightarrow j}$ and halt;
3. complete $V_i \cap V_j$ in $G_{i\rightarrow j}$ and denote resultant graph by $G'$;
4. create a JT $J'$ from $G'$;
5. create a JF $W_{i\rightarrow j}$ consisting of $J'$ and cliques in $L_{i\rightarrow j}$;
6. remove clique $V_i \cap V_j$ from $J'$ and $J'$ is broken into subtrees;
7. for each subtree of $J'$ rooted at clique $V_i \cap V_j$ with adjacent clique $C'$, do
8. find a clique $C$ from $L_{i\rightarrow j}$ such that $C' \cap C = C' \cap (V_i \cap V_j)$; /*with maximum intersection*/
9. if $C \subset C'$, remove $C$ from $W_{i\rightarrow j}$; else connect $C$ to $C'$;

From the triangulation, a junction forest (JF) is produced at each agent for each direction of inter-subnet message passing. The JT cliques (and hence linkages) are generally smaller than those in LJFs. Algorithm 20 shows how to build message JFs at each agent. The number of JT's included in a JF is upper bounded by the number of cliques in the corresponding interface and lower bounded by the number of subtrees rooted at the completed d-sepset. Algorithm 21 specifies how to construct an inference JT for intra-subnet local inference, where a locally moralized graph is triangulated based on fill-ins received from all neighbors in CommunicateFillin.
Algorithm 21 (BuildInferenceJT) (Xiang et al., 2006) Let $A_i$ be an agent with local moral graph $G_i = (V_i, E_i)$. Let adjacent agents of $A_i$ be $A_1^i, \ldots, A_m^i$, and $F_i^j$ be the set of fill-ins received by $A_i$ from $A_j^i$ during CommunicateFillin. When $A_i$ is called to BuildInferenceJT, it does the following:

1. eliminate $V_i$ from $G_i' = (V_i, E_i \cup \bigcup_{i=1}^m F_i^j)$;
2. add fill-ins obtained to $G_i'$;
3. construct a JT $J_i$ based on the resultant $G_i'$;

Algorithm 22 (FTCoBeliefUpdateLazyDLJF_CON) Each arbitrary agent $A_i$ with inference JT $J_i$ and a JF $W_i \rightarrow j$ for each neighbor $A_j^i$ ($0 \leq j < m_i$) performs the following operations to set up the environment and continuously propagate Lazy beliefs:

1. for $j = 0, \ldots, m_i - 1$, do
2. create a thread $T_{is}^j$ for sending messages to neighbor $A_j^i$;
3. create a thread $T_{ip}^j$ for receiving messages from neighbor $A_j^i$;
4. while (true)
5. synchronized(all linkage potentials to $J_i$) {
6. perform UnifyPotentialLazy using linkage potentials from all adjacent agents at $J_i$;}
7. sleep a random time;

Each $T_{is}^j$ performs the following operations:
8. while (true)
9. synchronized(all linkage potentials sent to $W_i \rightarrow j$) {
10. perform CollectPotentialLazy in each linkage host of $W_i \rightarrow j$;}
11. send each linkage potential to $A_j^i$ by SendPotentialLazy;
12. sleep a random time;

Each $T_{ip}^j$ performs the following operations:
13. while (true)
14. synchronized(all linkage potentials sent to $J_i$ and $W_i \rightarrow k$, $0 \leq k < m_i$, $k \neq j$) {
15. receive the linkage potentials sent from $W_j \rightarrow i$ at $A_j^i$;}

Belief propagation with DLJFs, like the extended Hugin inference or the extended Lazy inference on LJFs, consists of a round of inward propagation and a round of outward propagation on the hypertree. In belief propagation on DLJFs, a message is sent from a sending JF and is absorbed by a receiving JF. For a different neighbor, a different sending or receiving JF may be used. Between a pair of sending and receiving JFs, a linkage is created for each pair of sending and receiving cliques. The pair of sending and receiving cliques are called hosts of the linkage. SendPotentialLazy is used to send a linkage potential to an adjacent agent and before sending, CollectPotentialLazy should be called on the corresponding linkage host to ensure an updated belief. Note conditional probability tables (CPTs) are assigned to message JFs and inference JTs in the same way (see Definition 4 for belief assignment to JTs). On the same subnet, the beliefs of all JFs are identical and are equal to the belief of the inference JT.

With the generation of DLJFs formally presented and the related concepts formally introduced, we can provide a more formal answer to the problem discussed in the beginning of the section regarding
why Lazy division is not needed for message passing between agents. This is because message passing is done between JFs, and on each JT of a JF, there is one and only one linkage host. That is, linkage hosts and hence linkages are separated. There is no message duplication problem.

Algorithm 22 (FTCoBeliefUpdateLazyDLJF_CON) shows how Lazy extension on DLJFs can be further extended to infinite exact belief updating. Note a sending JF $W_{i \rightarrow j}$ at $A_i$ may have linkage potentials for the inference JT $J_j$ and JF $W_{j \rightarrow k} (0 \leq k < m_j, k \neq i)$ at $A_j$. Since a JF is neighbor and direction dependent, and the linkage potentials a JF can absorb are specifically those sent to it, both CollectPotentialLazy and SendPotentialLazy are called without specifying what linkage potentials to use or which neighbor to be relative to. For Algorithm 22, we have Proposition 3.

**Proposition 3** By Algorithm 22, all agents’ beliefs would converge to the exact ones.

Proof: With the similar argument as in the proof to Proposition 2, agents’ belief will converge within a time $O(d)$, where $d$ is the diameter of the hypertree. At agent $A_i$, the converged belief $B(V_i) = \prod_{C \in J_i} B(C) \prod_{Q} B(Q)$ is exactly the same as that obtained by the Lazy extension on DLJFs, where $B(C)$ is the product of potentials assigned to clique $C$ in inference JT $J_i$, and $B(Q)$ is the product of potentials assigned to linkage $Q$. Theorem 2 in (Xiang et al., 2006) indicates that the converged belief is exact. □

7 An Example

In this section, we demonstrate the approach with an iterated Lazy extension on LJFs example.

As shown in Figure 12 is the LJF of the MSBN in Figure 3, where each clique is assigned a set of initial probability tables (at time 0). For the convenience of analysis, we assume all the 5 respective agents are synchronized. Specifically, each agent sends one and only one message to each of its neighbors within one time unit and only the messages from the preceding time unit are used for the message and belief updating in the current time unit. Anyway, the asynchronous situations can be similarly analyzed. Then at time 1, messages $M_{0,1}, M_{2,1}$ and $M_{4,3}$ are guaranteed to be correct and all other messages may not be correct. Note although agents $A_0, A_2$ and $A_4$ can send correct messages at this time, they may not be able to form the correct beliefs since they have not received the messages that are guaranteed to be correct yet. At time 2, in addition to messages $M_{0,1}, M_{2,1}$ and $M_{4,3}$ which are still correct (any messages becoming correct would generally remain correct), messages $M_{1,3}$ and $M_{3,1}$ also become correct. Finally at time 3, all messages in the problem domain are correct. At time 4, all agents can safely form the correct beliefs.

To show how the particular messages are passed, we assume all variables are binary and for a variable $v$ without any parent, let

$$P(v = 0) = 0.6,$$

for a variable $v$ with single parent $p$, let

$$P(v = 0|p = 0) = 0.9, P(v = 0|p = 1) = 0.2,$$

and for a variable $v$ with two parents $p_1$ and $p_2$, let

$$P(v = 0|p_1p_2 = 0) = 0.1, P(v = 0|p_1p_2 = 1) = 0.8,$$

$$P(v = 0|p_1p_2 = 2) = 0.6, P(v = 0|p_1p_2 = 3) = 0.3.$$
The orders of the two parents are respectively \( P(d|h,v), P(e|y,x), P(v|y,b), P(a|z,u) \) and \( P(j|u,b) \) for all the relevant variables. We focus on examining how message \( M_{3,4} \) and the belief of agent \( A_1 \) evolve. All other messages and agents’ beliefs can be similarly analyzed. In the analysis, we use \( B(s) \) to denote the potential of the set \( s \) of variables.

![Figure 12: A LJF for the MSBN in Figure 3: 5 local JTs (J0 - J4) and 4 linkage trees (L0 - L3). Over each linkage tree there is one message for each direction, and the message from \( J_i \) to \( J_j \) is denoted by \( M_{i,j} \). Next to each clique is the set of probability tables assigned to it. Each thick link relates the linkage to its host in a local JT.](image)

Since at time 0, all messages are composed of the uniform probability distributions, at time 1 we can get

\[
M_{3,4}^1 = B^1(h, d) = \{0.99, 1.21, 0.81, 0.99\}
\]

for \( h,d=0,1,2,3 \), respectively (the superscript ‘1’ indicates time 1). This is obtained from clique \( d,h,v \) after a message passing from clique \( b,h,v \) to \( d,h,v \) in \( J_3 \). If agent \( A_4 \) calculates its marginal belief about \( h, g, d \) based on \( M_{3,4}^1 \), it would get

\[
P^1(h) = \{0.55, 0.45\}, P^1(d) = \{0.45, 0.55\}, P^1(g) = \{0.515, 0.485\}
\]

for \( h = d = g = 0 \) and 1 respectively. This is not right since it is different from the expected

\[
P^e(h) = \{0.585, 0.415\}, P^e(d) = \{0.5205, 0.4795\}, P^e(g) = \{0.56435, 0.43565\}
\]

The reason is that message \( M_{3,4} \) has not converged to the right one yet at time 1.

Since messages \( M_{0,1}^1 \) and \( M_{2,1}^1 \) are relevant for the calculation of correct message \( M_{1,3}^2 \), we take a closer look at them. At time 1, the Lazy message \( M_{0,1}^1 \) contains \( P^1(z) \) and \( P^1(u) \) over linkage \( \{u,z\} \) and \( P^1(u, b) \) over linkage \( \{b,u\} \), respectively. However, only \( P^1(z) = \{0.6, 0.4\} \) is effective since the others are uniform probability distributions. \( M_{2,1}^1 \) is also a uniform probability distribution.

At time 2, since all messages needed (i.e., \( M_{0,1}^1 \) and \( M_{2,1}^1 \)) for agent \( A_1 \) to send the right message to agent \( A_3 \) are ready, we can get the right message \( M_{1,3}^2 \). Note since \( M_{1,3}^1 \) is not guaranteed to be correct, we omit its analysis to save space. By a Lazy belief propagation in JT \( J_1 \) based on its initial belief assignment and the effective part of message \( M_{0,1}^1 \), we can get message

\[
M_{1,3}^2 = B^2(b, v) = \{0.165, 0.385, 0.27, 0.18\}.
\]

At time 3, agent \( A_3 \) is ready to send the right message \( M_{3,4}^3 \) to \( A_4 \). By absorbing \( M_{1,3}^2 \), we get

\[
P(h, v) = \{0.2025, 0.3825, 0.2325, 0.1825\}
\]
from clique \{b,h,v\} at \(J_3\). Then from clique \{d,h,v\} we can get

\[ M_{3,4}^3 = B^3(h, d) = \{0.32625, 0.25875, 0.19425, 0.22075\} \]

after a message passing from clique \{b,h,v\} to clique \{d,h,v\}.

Finally, at time 4, agent \(A_4\) can safely calculate

\[ P^4(h) = \{0.585, 0.415\}, P^4(d) = \{0.5205, 0.4795\} \]

from clique \{h,d\}, and

\[ P^4(g) = \{0.56435, 0.43565\} \]

from clique \{g,d\} after a message passing from clique \{h,d\} to clique \{g,d\}. They are exactly the same as the expected.

Note the above analysis is all about the prior belief propagation, but the posterior belief analysis based on the propagation of the effect of the evidence can be similarly performed since the observations would not change the nature of the probability distribution tables but make them smaller.

8 Discussion

In this section, we discuss other properties of the multiagent infinite belief propagation proposed in this paper, especially those related with adverse environments.

8.1 Fault-Tolerance

As mentioned in Section 4, in this paper, by “fault-tolerance” we mean (1) the eventual convergence of agents’ beliefs to the correct ones would not be prevented by temporary problems; and (2) for permanently disconnected MSBNs, iterated belief updating would make all agents within each connected portion belief consistent disregarding any temporary problems, and the consistent belief at each portion is locally correct relative to the initial belief of the portion when the portion is first formed (by disconnection). Such consistent beliefs may not be globally correct, however. This is because each portion may not have acquired proper belief from other portions when disconnection occurs. For a unified description, we can say, for each connected portion or the whole connected system, the consistent belief reached by iterated belief updating disregarding any temporary problems would be correct relative to its initial belief assignment. Such fault-tolerance would lead to “error-corrective” in the sense that any errors introduced by the temporary (communication, computing, etc) problems would be corrected by eventually properly delivered (calculated) beliefs.

8.2 Scalability

When an agent from a system is allowed to create a pair of receiving and sending threads to accommodate a joining request from a different system, the system becomes scalable. This can be done by a slight change to Algorithm 16: when any joining request is detected by an agent, a pair of threads is created to establish connection with the corresponding threads and propagate potentials. All previous work on LJFs or DLJFs are not scalable.
8.3 Converging Speed

Let the diameter (the number of hyperlinks of a longest path) of the hypertree of an MSBN be \( d \), and the maximum time needed for a pair of adjacent agents to finish a message exchanging over their \( d \)-sepsset be 1. As discussed in the proof for Proposition 2, under normal conditions, all linkage potentials would converge to the correct beliefs by at most a time \( d \). Since the belief of an agent is the combination of all its linkage beliefs and its own initial belief, the agent’s belief would converge to the right one in the meantime. This conclusion also applies to the iterated Shenoy-Shafer extension and the iterated Lazy extension on DLJFs. Under abnormal conditions, the time needed for convergence may be highly dominated by the lasting time of the temporary problems (faults). Before the temporary problems (faults) disappear, belief convergence may not be possible. In either normal or abnormal condition, the number of iterations may not be a good measure for converging speed since some agents may work faster (based on faster machines) and hence iterate more.

![Diagram of agents and messages](https://via.placeholder.com/150)

Figure 13: At time \( t \), the impacts of the last few messages may co-exist among a set of connected agents, where \( A_0 \) is the failed agent, \( L_i \) (\( 1 \leq i \leq n \)) is, relative to time \( t \), the \( i \)'th to the last message from \( A_0 \) to \( A_1 \), and \( E_{L_i} \) is the message that contains the effect of \( L_i \). The effects of the last \( n \) messages from \( A_0 \) to \( A_1 \) would be distributed as shown when all agents iterate in the same rate.

8.4 Error from Failing Agents

Permanently failed agents may introduce errors to the beliefs of other agents. Depending on the causes of the failures, the failed agents may or may not produce any messages, and may or may not be connected with other agents. If the failed agents are disconnected from others or no more messages are sent out from them, the influence of the failed agents on other agents is limited to the last messages they sent out (before disconnection if disconnected) and received by other agents. Any such last message received by an agent would be propagated to all relevant agents that have an active path with it. If the message is wrong, it would contribute to the errors in agents’ beliefs. If the failed agents are connected with other agents and keep sending out messages, the influence of the failed agents on other agents would vary depending on the characteristics of the last few messages received by other agents. This time it is the last few messages instead of the last messages that would influence the beliefs of other agents because the effects of the last few messages may co-exist in the relevant agents’ beliefs, and in particular such last few messages may keep changing. For example, as shown in Figure 13, at different time \( t \), the set of last \( n \) messages from \( A_0 \) may vary. In particular, the message sent from \( A_0 \) to \( A_1 \) may never converge due to the failure of \( A_0 \). If the message from \( A_0 \) does not converge, the beliefs of all affected agents would not converge, either. Regarding the level of errors introduced by the failing agents to the converged beliefs, it would vary greatly depending on the failing message absorbed, its difference with the right message, and the strength of the dependency of agents’ beliefs on such messages. There is generally no good way to effectively measure the level of such errors though we can always estimate the difference between the right belief distribution and the wrong belief distribution by relative entropy or KL divergence (Cover & Thomas, 1991).
8.5 Infinite Belief Updating

With the iterated multiagent belief updating proposed in the paper, all agents update their beliefs infinitely. By infinite belief updating, all agents’ beliefs converge to the rights ones (relative to their initial belief assignment) whenever possible. However, it may waste computing resources since belief updating would continue even after beliefs’ convergence. We can address this issue by belief consistency checking from time to time. By belief consistency checking, belief updating would stop whenever agents’ beliefs are globally consistent (Xiang, 2000), and whenever necessary (e.g., new evidence is entered), another round of iterated belief updating could be started. Agents’ beliefs are globally consistent if every pair of adjacent agents have consistent beliefs on their d-sepset (shared variables). Belief consistency checking can be done by one agent who from time to time checks the belief consistency of every pair of neighboring agents or by one party of every pair of adjacent agents who receives the belief of the other party on their d-sepset and compare it with its own belief. We may set a time-out period for iterated belief updating so that we may not wait its completion by endlessly checking belief consistency in case permanent disconnection occurs or lasting faults exist. Belief consistency checking may need to be synchronized with other belief updating operations.

9 Conclusion

Hugin, Shenoy-Shafer, and Lazy propagation architectures on JTIs have all been extended to LJFs and/or DLJFs for multiagent belief updating, but these extensions might not be suitable for practical multiagent environments. Belief updating based on these extensions is vulnerable to any problems or errors occurred at agents or communication channels. In this paper, we propose to further extend these extensions for robust multiagent belief propagation.

The further extensions are made based on (infinitely) iterative belief updating. Issues related with the further extensions are discussed and solutions are presented. We first present a general iterative belief updating algorithm that can effectively prevent multiagent or multithread deadlocks, where agents autonomously and asynchronously keep updating their beliefs. We then show that Hugin or Lauritzen-Spiegelhalter extensions on LJFs might not be able to be further extended to support iterated exact belief propagation. We finally show that Shenoy-Shafer and Lazy extensions on LJFs or DLJFs can be further extended to support infinite iterated coherent belief updating. The computational complexity of each viable further extension is analyzed and shown to be linear in the diameter of the hypertree under normal conditions.

The proposed belief updating approach is scalable while performing belief propagation, which makes multiagent agent interpretation systems real-time open for any joining requests of other systems. The infinite iterative belief updating can be made finite by from time to time belief consistency checking of each pair of agents to detect global belief consistency among all agents.

The performance of the proposed approach under abnormal conditions is discussed. The approach is fault-tolerant in the sense that the beliefs of all connected agents (not permanently disconnected) would converge to the right ones, relative to their initial beliefs, disregarding any temporary problems or faults. This makes the approach robust in the adverse multiagent environments.

4 The inter-agent message computation and exchanging complexity for each further extension is the same as that for Shenoy-Shafer or Lazy extensions on LJFs or DLJFs. Such computational complexity is $O(c^2q)$, where $c$ is the maximum number of clusters (cliques) in an inference JT or a message JF and $q$ is the cardinality of the largest cluster (clique).
References


