ABSTRACT

ITERATIVE AGGREGATION OF BAYESIAN NETWORKS
INCORPORATING PRIOR KNOWLEDGE

by Jian Xu

Multi-source information integration has gained significant interest recently. We focus on integrating Bayesian networks (BNs) learned from data. The BN batch aggregation algorithm proposed by Maynard-Reid II and Chajewska (MC01) requires all sources’ information be available at aggregation time and does not take the user’s prior knowledge into account. We extend this algorithm to make the aggregation iterative, supporting “anytime” querying, and to allow the incorporation of the user’s structural prior knowledge. We prove that the iterative extensions for joint distribution aggregation are independent of the order in which sources arrive. We show experimentally that iterative BN aggregation is order-dependent due to bias introduced by the algorithm’s optimization nature and an “inertial” effect. However, we show that the results compare well with the batch algorithm in accuracy and efficiency. We also show experimentally that incorporating the user’s structural prior knowledge can improve the accuracy and efficiency.
ITERATIVE AGGREGATION OF BAYESIAN NETWORKS
INCORPORATING PRIOR KNOWLEDGE

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Systems Analysis
Department of Computer Science & Systems Analysis

By Jian Xu

Miami University
Oxford, Ohio
2004

Advisor ______________________
Dr. Pedrito Maynard-Zhang

Reader _______________________
Dr. Mufit Ozden

Reader _______________________
Dr. Valerie Cross
TABLE OF CONTENTS

1. Introduction ............................................................................................................................1
2. Information Integration Overview ..........................................................................................3
3. Probabilistic Sources ..............................................................................................................5
   3.1 Joint distribution ...........................................................................................................5
   3.2 What is a Bayesian network? .......................................................................................6
   3.3 Why are Bayesian networks important? .......................................................................8
   3.4 BN inference .................................................................................................................9
   3.5 Probabilistic learning ..................................................................................................11
4. Batch Aggregation Algorithm ..............................................................................................14
   4.1 Aggregation of joint distributions ..............................................................................14
   4.2 Aggregation of Bayesian networks ............................................................................17
5. Iterative Aggregation ............................................................................................................19
   5.1 Iterative aggregation algorithms .................................................................................19
   5.2 Correctness ....................................................................................................................21
   5.3 Properties ....................................................................................................................22
6. BN Aggregation with Structural Prior Knowledge ..............................................................24
   6.1 PriDAG aggregation algorithm ..................................................................................24
   6.2 Properties ....................................................................................................................25
7. Experimental Results ............................................................................................................27
   7.1 Results of iterative aggregation ..................................................................................28
      7.1.1 Comparison of accuracy to sources and batch DM .............................................28
      7.1.2 Aggregate source bias and ‘inertia’ .................................................................30
      7.1.3 Sensitivity of accuracy to order ........................................................................35
      7.1.4 Time cost .............................................................................................................38
   7.2 Results of aggregation with PriDAG ..........................................................................39
      7.2.1 Time saving .........................................................................................................40
      7.2.2 Accuracy comparison .........................................................................................42
8. Conclusion and Future Work ...............................................................................................43
References ................................................................................................................................45
Appendix ..................................................................................................................................48
   A. PROOFS ......................................................................................................................48
   B. CODE .........................................................................................................................51
LIST OF TABLES

Table 1: An example of a joint distribution.................................................................5
LIST OF FIGURES

Figure 1: Cancer BN Example ...............................................................................................7
Figure 2: MDL BN learning algorithm ..................................................................................13
Figure 3: Brute force iterative aggregation algorithm ..........................................................19
Figure 4: Iterative BN aggregation algorithm ......................................................................20
Figure 5: Modified MDL BN learning algorithm that satisfying given structural constraints. 25
Figure 6: Comparison of iterative, batch, and source average accuracy over time when iteratively combining same-size sources trained on samples from the Asia network ..........29
Figure 7: Comparison of iterative, batch, and source average accuracy over time when iteratively combining same-size sources trained on samples from the Cancer network .......29
Figure 8: Comparison of average iterative and batch accuracy over time when iteratively combining same-size sources trained on varied number of samples from the Asia network...30
Figure 9: Illustration of how inertial bias can be counteracted when sources of weight and size equal to the aggregate agent’s are integrated with it. ......................................................31
Figure 10: Illustration of how inertial bias can be counteracted even more when sources of weight and size twice as large as the aggregate agent’s are integrated with it..........................31
Figure 11: Effect on accuracy when a highly weighted, inaccurate source arrives before 10 sources of lower weight and accuracy ..............................................................................32
Figure 12: Same as Figure 11, except the highly weighted source arrives midway between the arrivals of the 10 low-weight sources ........................................................................33
Figure 13: Same as Figure 11, except the highly weighted source arrives after the 10 low-weight sources ..................................................................................................................33
Figure 14: Effect on accuracy when a highly weighted, accurate source arrives before 10 sources of lower weight and accuracy ..............................................................................34
Figure 15: Same as Figure 14, except the highly weighted source arrives midway between the arrivals of the 10 low-weight sources ........................................................................34
Figure 16: Same as Figure 14, except the highly weighted source arrives after the 10 low-weight sources ..................................................................................................................34
Figure 17: Variation in aggregate accuracy given random ordering of same-size sources......35
Figure 18: Variation in aggregate accuracy given different random ordering of same-size sources .................................................................................................................................36
Figure 19: Variation in aggregate accuracy given increasing KL-divergence order of same-size sources .................................................................................................................................36
Figure 20: Comparison of total aggregation time of iterative and batch aggregation algorithms as the number of sources increases from 1-15 for fixed source sizes 20, 50, 100, and 1000...38
Figure 21: Comparison of number of BNs visited when increasing the number of nonfixed edges in the PriDAG when aggregating 20 sources seeing 50 samples each .............................40
Figure 22: Comparison of time spent on aggregations when increasing the number of nonfixed edges in the aggregation of 20 sources seeing 50 samples each .......................................40
Figure 23: Comparison of time spent on optimal aggregation when increasing the number of nonfixed edges in the aggregation of 20 sources seeing 50 samples each ...................................41
Figure 24: Comparison of accuracy of aggregate models when increasing the number of nonfixed edges in the aggregation of 20 sources with 50 samples each .....................................42
ACKNOWLEDGMENTS

I would like to thank Prof. Pedrito Maynard-Zhang, my thesis advisor, for his constant support, valuable advice and encouragement. I also want to thank my thesis committee members, Dr. Ozden and Dr. Cross, for their guidance, and their time in reviewing this work.
1. Introduction

The explosion of available information via the Internet has drawn significant interest in both the database and artificial intelligence communities to the problem of integrating information from multiple sources. Whereas the database community has focused on the problem of querying distributed databases (DDH01, Sujansky01, KK03), the artificial intelligence community has focused on the problem of combining domain models (MA92, PW99, MC01, RD03, NM00). In fact, important and interesting problems lie on the interface of the two realms. For example, information sources may not always be able to provide their data, but may be able to provide a summative model generated from the data. We are interested in the problem of information integration (also known as “aggregation”, “combination” and “fusion”) in stochastic domains where the information that sources send can be modeled as a probabilistic distribution learned from data.

Consider the following motivating scenario: A medical intern is diagnosing a patient. In making her diagnosis, she wants to take into account not only the patient’s symptoms, history, and test results, but also the accumulated knowledge of more experienced doctors around the country. Unfortunately, the doctors cannot provide the patient data on which their experience is based due to privacy concerns, bandwidth concerns, or simple data unavailability (old records may have been discarded). However, they can provide probabilistic models – either via elicitation or application of a learning algorithm – that encapsulate their experience and beliefs to date. These models describe, for example, the probabilistic relationship between diseases and symptoms. Thus, we want to provide the intern with an expert system that accepts these probabilistic models, and combines them dynamically as they arrive so as to provide her queries with the best information available at any time; we also want the aggregation system to allow the intern to input her prior knowledge about this domain, and have it represented in the final combined expert system. We would hope that (1) this aggregation process would take into account information about the doctors such as years of experience and areas of expertise, especially in the likely case that doctors disagree, (2) the final expert system after all the doctors’ models arrive does not depend on the order in which
they arrive, and (3) the final expert systems encodes the user’s prior knowledge in the form of causal relationships among the domain variables.

We focus on those cases where each model is a Bayesian network (BN) learned using the Minimum Description Length (MDL) algorithm. A BN is a popular, compact, graphical encoding of a probability distribution. Very little work has been done in the area of aggregating distributions learned from data, especially when they are represented as BNs. Maynard-Reid II and Chajewska (MC01) have proposed aggregation algorithms for such situations, but these algorithms are batch in nature, and without much user involvement. Thus, we extend this batch aggregation algorithm along two dimensions: (1) to make it iterative and, hence, “anytime”, and (2) to allow the incorporation of structural prior knowledge from the user. The expectation is that the extensions will also improve the performance of the algorithm, making it more practical, adaptable and satisfying to users. We then validate these two extensions experimentally using data generated from well-known medical diagnosis BNs existing in the literature and compare our results to the original batch algorithm. The comparison shows us that our algorithms are more adaptable and efficient in practice, and even more accurate in the algorithm incorporating prior knowledge (called the PriDAG algorithm in this paper), but with only little loss of accuracy in the iterative aggregation algorithm.

The rest of the paper is organized as follows: In Sections 2, we provide a brief overview on information integration. In Section 3, we provide brief background on probabilistic models focusing on the issues related to Bayesian networks. In Section 4, we introduce the batch aggregation algorithm. In Section 5, we introduce the iterative algorithm for joint distributions, extend it to BNs, and describe its properties. In Section 6, we introduce the PriDAG algorithm for BNs, and describe its properties. In Section 7, we describe experiments comparing the accuracy and efficiency of the algorithms with that of the original batch algorithm. Finally, in Section 8, we conclude with a discussion of the algorithms’ weaknesses and possible extensions.
2. Information Integration Overview

Information integration has received significant attention in both the AI and database communities as evidenced by numerous research projects, for example, the TransFER project which uses the Standard Query Model Framework (SA94), and the PROMPT project which is a semi-automatic algorithm and tool for ontology merging and alignment (NM00). Effective integration of heterogeneous databases and information sources has been cited as one of the most pressing challenges in domains ranging from corporate data management to bioinformatics (KK03). Information integration is also common in complex decision domains such as medical diagnosis.

In this research, we focus on the integration of information that is inherently uncertain. Thus, we consider expert information to be represented in terms of probability models. Probability models have been very important in AI because uncertainty is ubiquitous and is often represented with probability distributions. Different experts may give different probability models according to his/her own experience in a certain area. How to integrate these probability models and draw reasonable conclusions becomes a big problem in such scenarios. Although there has been a lot of research done in information integration, many issues remain without complete solutions. We list some of the major ones related to our research:

(1) Resolving ontological differences: Specifically, there exist four types of differences: structural differences, naming differences, semantic differences, and content differences. For example, naming differences occur when using distinct lexical terms to name the same semantic object across sources, such as “doctor” versus “physician”; semantic differences occur when the labels of tables, fields and data values are similar or even identical but with different meanings, for example, “doctor” can be used in both medical and scientific area but with different meanings. There are a lot of database people dealing with this problem (DDH2003). In our approach, we assume every source has the same ontology.

(2) Detecting and resolving conflicts: the information coming from different sources may not be consistent, so detecting the conflicts between them and resolving these conflicts is an
important issue. Until now, most of the conflicts are resolved manually. Our work endeavors to resolve the conflicts automatically.

(3) Efficient inference and optimization: The main goal of information integration is to pose a query over the distributed heterogeneous sources and get the results efficiently and accurately. Several important choices result in tradeoffs between these goals. For example, for each query in database integration, we may go to the real location of each source to get the results and then combine them, or we may integrate all the information of all the sources and store the aggregated information in a central place, then query over the aggregated information (Gardner98). Whereas the former strategy will absolutely save space at the expense of more time and heavier network traffic, the latter strategy may save time and get the result of a query faster but use more space while making it harder to get the most up-to-date information if the sources keep changing. In our research, the main goal is to efficiently inference over the source probability models (BNs) to get an accurate result. Inspired by the second approach used in the database field, the approach we use is to combine all the information from different source BNs and only store the aggregate BN (meta information) at a central place for inference and assume the sources are stable.
3. Probabilistic Sources

Because uncertainty is represented in terms of probabilities, we consider expert information in terms of probability distributions. We briefly introduce two types of probability models—joint distribution and BN—and the associated inference and learning algorithms, with emphasis on Bayesian networks. For more details, see (Murphy01).

3.1 Joint distribution

A joint distribution is the simplest representation of a probability distribution. For discrete domains, it is essentially a table that specifies the probability of every possible instantiation of the variables. Suppose that a medical cancer domain consists of three binary variables $M$, $S$, $B$, which represent Metastatic Cancer (present or absent), Serum Calcium (increase or decrease), and Brain Tumor (present or absent), respectively. The joint can be represented by a joint probability table as shown below. For example, the joint probability of this domain where $M =$ present (+), $S =$ decrease (-), $B =$ absent (+) would be $P(M, \neg S, B) = 0.008$.

<table>
<thead>
<tr>
<th>$M$</th>
<th>$S$</th>
<th>$B$</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0.032</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>-</td>
<td>0.128</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>+</td>
<td>0.008</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>-</td>
<td>0.032</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>+</td>
<td>0.008</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>-</td>
<td>0.152</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0.032</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.608</td>
</tr>
</tbody>
</table>

Table 1: An example of a joint distribution
Probabilistic inference involves computing the posterior probabilities of query propositions, possibly conditioned on values for some evidence variables. The straightforward algorithm for computing marginals is to enumerate and sum the probabilities of all the entries consistent with the values of the query variable. Conditional probabilities are simply the division of two marginals. For example, to find the probability of $S = \text{increase (+)}$ when $B = \text{present (+)}$:

$$P(S = + | B = +) = P(S = +, B = +) / P(S = +)$$

$$= \sum_m P(m, S = +, B = +) / \sum_{m,b} P(m, S = +, b)$$

$$= (0.032 + 0.008) / (0.032 + 0.008 + 0.008 + 0.152) = 0.2$$

Unfortunately, the joint distribution representation does not scale well for large and complex domains. The table requires space exponential in the number of variables (e.g., $2^n - 1$ for $n$ binary variables). Additionally, the estimation of so many numbers becomes impractical, and inference requires exponential time as well (e.g., computing the marginal on one variable requires accessing $2^{n-1}$ parameters for $n$ binary variables).

Because of the drawbacks associated with joint distributions, Bayesian networks (BNs) have become a preferred way of representing probability distributions due to their many attractive properties. Consequently, we use BNs to represent the probability distributions of the experts. We now give a brief overview of BNs. For more detailed information, see (Heckerman96).

### 3.2 What is a Bayesian network?

A Bayesian network (BN), also known as a causal probabilistic network, is a compact graphical representation of a probability distribution over a set of domain random variables $X = \{X_1, X_2, ..., X_n\}$. It consists of a structural part and a quantitative part.

1. Structural (or qualitative) part: The structure of a BN is a directed acyclic graph (DAG) $S$ over a set of nodes, where each node represents a unique random variable $X_i$. This structure graphically encodes a set of conditional independence relations among the variables in $X$. In practice, the edges in the graph are typically interpreted as cause-effect relationships, especially for BNs elicited from experts. We use $X_i$ to denote both the variable and its corresponding node, and $Pa_i$ to denote the parents of node $X_i$ in $S$ as well as the variables corresponding to these parents.
(2) Numerical (or quantitative) part: Associated with each node is a local conditional probability distribution (CPD) giving the probability of each value of the variable given each value combination of the parents. The parameters in the CPDs make up the numerical part of the BN.

The graphical structure and CPDs together are sufficient to represent the joint probability distribution for $X$. In particular, given a particular BN, the joint probability distribution for an instantiation $X$ of the set of $n$ variables $X$ is given by:

$$p(X) = \prod_{i=1}^{n} p(x_i \mid pa_i)$$

where $x_i$ is the $i$th variable’s value in $X$ and $pa_i$ are the values of the $i$th variable’s parents in $X$.

A simple Bayesian network from the literature for a Cancer medical domain is shown in Figure 1.

![Figure 1: Cancer BN Example](image)

The above network displays the causal relationships among five domain variables {Metastatic Cancer (M), Serum Calcium (S), Brain Tumor (B), Coma (C), Headaches (H)}. Each of them has two values: increase or decrease for Serum Calcium, and present or absent for the others.
The arcs in the structure represent the direct causal relationships between two variables, such as metastatic cancer causes brain tumor, and brain tumor causes coma and headaches, etc. The CPDs specify the effects the parent nodes have on the child node, for example, if serum calcium increases and brain tumor exists, the probability of a coma is 0.80, i.e., $P(C = + | S = +, B = +) = 0.80$. The BN structure also graphically encodes the conditional independences among domain variables, where two variables are conditionally independent of each other if all paths of influences are blocked. An example in the Cancer BN is that given brain tumor, metastatic cancer and headaches are independent of each other; however, this is not the case if given nothing.

3.3 Why are Bayesian networks important?

Bayesian networks have become popular as a representation for encoding uncertain expert knowledge in expert systems (Heckerman96). The main reason that BNs are important is their theoretically sound foundation in probability and graph theory, their usefulness in diverse contexts, and their proven effectiveness in applications. In particular, they are compact, easy to elicit and interpret, encode causal relations and conditional independences, support efficient inference, and can be learned from data.

(1) Representation: Using a BN to graphically represent a probability distribution is much more compact. In general, if we have $n$ binary random variables in a domain, the full joint distribution would require $O(2^n)$ parameters, whereas using a BN typically significantly reduces the number of parameters ($O(n2^k)$ for binary variables with a maximum of $k$ parents each). For example, in the above Cancer BN, $2^5 - 1 = 31$ parameters are needed to represent the joint distribution, but only 11 parameters are needed if we use a BN. Furthermore, the structure of a BN encodes the causal relations in the domain and the conditional independences among domain variables.

(2) Elicitation and interpretation: A BN is very easy to elicit and interpret. To elicit a BN from an expert in a domain, we can easily get it by asking the expert questions which are generally easy for the expert to answer: What are the relevant domain variables and what are their values? What are the direct causal relationships among the variables? What is the probability of each value of each variable given the different possible instantiations of the variables that
have a direct causal impact on it? Moreover, the causal relationships can be used to gain understanding about a problem domain, and therefore make the interpretation much easier.

(3) Inference: Inference in many real-world Bayesian networks has proven to be tractable. By taking advantage of the compact structure of the BN and the explicit conditional independences among variables, the inference can get around the exponential blowup of the joint distribution and can be quite fast, though we cannot improve it in the worst case. Many BN inference algorithms have been developed to routinely perform inference, from exact to approximate, such as the variable elimination algorithm, the join-tree algorithm, and Monte Carlo sampling-based approximation algorithms. In our aggregation algorithm, we use the join-tree inference algorithm. We introduce the variable elimination and the join-tree algorithms in more detail in Section 3.4.

(4) BNs can also be learned from data using some now standard learning algorithms, such as the search-based Minimum Description Length learning algorithm (described in Section 3.5). Bayesian methods in conjunction with BNs and methods from information theory offer an efficient and principled approach for avoiding the overfitting of data. Prior knowledge and data can be seamlessly combined in the BNs. In addition, BNs readily handle incomplete data sets. Since the Bayesian model encodes dependencies among all variables, it readily handles situations where some data entries are missing or one of the inputs is not observed (H1996).

3.4 BN inference

As mentioned in the above section, there exist many different inference algorithms in Bayesian networks, both exact and approximate. The variable elimination algorithm is one of the simplest exact algorithms. It uses independences to write a given query as a sum of products of terms found in the CPDs, pushes in the marginalization sums so as to minimize redundant computation, and iteratively marginalizes out the variables not appearing in the query. For this simple algorithm, we show an example here to give some intuition of BN inference. In the above Cancer BN, suppose we want to find the probability that there exists metastatic cancer given a coma has occurred, i.e. \( P(M = + | C = +) \).
From the definition of conditional probability, we know that

\[ P(M = + \mid C = +) = \frac{P(M = +, C = +)}{P(C = +)} \]

so we calculate \( P(M = +, C = +) \) first.

\[
P(M = +, C = +) = \sum_{s,b,h} P(M = +, s, b, h, C = +) \\
= \sum_{s,b,h} P(h \mid M = +, s, b, C = +)P(M = +, s, b, C = +) \quad \text{(Product rule)} \\
= \sum_{s,b,h} P(h \mid b)P(M = +, s, b, C = +) \quad \text{(Independence)} \\
= \sum_{s,b,h} P(h \mid b)P(C = + \mid M = +, s, b)P(M = +, s, b) \\
= \sum_{s,b,h} P(h \mid b)P(C = + \mid s, b)P(s \mid M = +)P(M = +, b) \\
= \sum_{s,b,h} P(h \mid b)P(C = + \mid s, b)P(s \mid M = +)P(b \mid M = +)P(M = +) \\
= P(M = +) \sum_{b} P(b \mid M = +) \sum_{s} P(s \mid M = +)P(C = + \mid s, b) \sum_{h} P(h \mid b) \quad \text{(Reorder terms and “push in” sums)} \\
= P(M = +) \sum_{b} P(b \mid M = +) \sum_{s} P(s \mid M = +)P(C = + \mid s, b)(0.8 + 0.2) \quad \text{(Marginalize out h)} \\
= P(M = +) \sum_{b} P(b \mid M = +) \sum_{s} P(s \mid M = +)P(C = + \mid s, b) \\
= P(M = +) \sum_{b} P(b \mid M = +) \{b = + : 0.8*0.8+0.2*0.8, b = - : 0.8*0.8+0.2*0.05\} \\
= P(M = +)0.2*0.8+0.05*0.74 = 0.2*0.197 = 0.0394
\]

\( P(C = +) \) is a normalization constant, i.e., we can compute it by repeating the above calculations with \( M = - \) to find \( P(M = -, C = +) \), then adding the two results together:

\[
P(M = -, C = +) = P(M = -) \sum_{b} P(b \mid M = -) \sum_{s} P(s \mid M = -)P(C = + \mid s, b)P(h \mid b) \\
= P(M = -) \sum_{b} P(b \mid M = -) \sum_{s} P(s \mid M = -)P(C = + \mid s, b) \\
= P(M = -) \sum_{b} P(b \mid M = -) \{b = + : 0.2*0.8+0.8*0.8, b = - : 0.2*0.8+0.8*0.05\} \\
= P(M = -)0.05*0.8+0.95*0.74 = 0.8*0.743 = 0.5944
\]

Thus, \( P(M = + \mid C = +) = \frac{P(M = +, C = +)}{P(C = +)} = \frac{0.0394}{0.0394 + 0.5944} = 0.0622 \)

In our work, we have chosen the join-tree inference algorithm, another exact inference algorithm, which is a little more complicated but much more efficient than the variable elimination algorithm. The join-tree algorithm transforms the BN into a polytree (i.e., there are no cycles in the equivalent undirected graph) by joining individual nodes to form cluster nodes. Inference in polytrees can be efficiently accomplished using a form of constraint
propagation where the constraints ensure that neighboring clusters agree on the posterior probability of any variables that they have in common. If the cluster sizes are small, then this algorithm is extremely efficient – linear time in the number of nodes if the network is already a polytree.

3.5 Probabilistic learning

Learning involves estimating a process that has generated a set of data by combining the data with a space of hypotheses modeling the process to predict future data.

In stochastic domains, the process we want to estimate is a probability distribution over a set of domain variables and the hypothesis space is a (possibly parameterized) set of probability distributions in the chosen representation (e.g., the set of all BNs).

From a Bayesian perspective, the optimal solution to this problem is to put a prior over the space of hypotheses and then use all the hypotheses weighted by their posterior probabilities given the data to do prediction. Thus, for example, the probability of some query $Q$ given some evidence $E$, data $D$, and hypothesis space $H$ would be

$$p(Q \mid E, D) = \int_{h \in H} p_h (Q \mid E) p(h \mid D) dh$$

where $p_h$ is the distribution corresponding to $h$. $p(h \mid D) = \alpha p(D \mid h) p(h)$ is the posterior probability of $h$ where $p(D \mid h)$ is the likelihood of the data given the hypothesis, $p(h)$ is the prior probability of $h$, and $\alpha$ is a normalization constant.

The problem with this approach is that most hypothesis spaces consisting of BNs are infinite so that doing the computation above is intractable at best, impossible at worst. As a result, we typically attempt to approximate the optimal approach by using the data to choose one “best” hypothesis. This is called model selection.

Different criteria can be used to define the “best” hypothesis. A common criterion is to pick the hypothesis with the maximum posterior probability given the data. This approach to learning is called maximum a posteriori estimation (MAP).
Definition 1: If $H$ is a hypothesis space of BNs over $n$ random variables, then the MAP BN over random variables given data set $D$ and prior $p(H)$ is the BN

$$\text{MAP}(p(H), D) \equiv \arg \max_{h \in H} p(h \mid D) = \arg \max_{h \in H} p(D \mid h) p(h)$$

Although this definition is still over an infinite space, for many important special cases such as independently and identically distributed (IID) data and Dirichlet priors, it is possible to analytically solve for the MAP hypothesis rather than explicitly search the space.

It is often difficult or impossible to specify a prior over the hypothesis space, so we often make the assumption that it is uniform. This reduces the MAP to finding the BN with the maximum likelihood to have produced the data. This process is called maximum likelihood estimation (MLE).

Definition 2: If $H$ is a hypothesis space of BNs over $n$ random variables, then the MLE BN over random variables given data set $D$ is the BN

$$\text{MLE}(D) \equiv \arg \max_{h \in H} p(D \mid h)$$

It can be easily shown that the CPD parameters that maximize likelihood given a particular DAG are easy to find. Thus, it suffices to find the DAG which, when parameterized with the MLE parameters, maximizes the likelihood. Unfortunately, this task has been shown to be NP-hard in general. Thus, it is common to resort to heuristic optimization search through the space of networks. Random restarts can be added to avoid local optima in the search space.

A second problem with using MLE for BN learning is that the MLE BN will always be the fully connected BN. This eliminates all the benefits of the potential compactness of the BN representation. Furthermore, it represents a case of overfitting, i.e., an inability to generalize due to “memorization” of the data.

One way to address this is to add a penalty term to each network’s score for network complexity. One such popular score is the minimum description length (MDL) score.

Definition 3: The MDL score of BN $b$ with an N-node DAG $g$ given data $D$ of size $M$ is

$$\text{Score}_{\text{MDL}}(b : D) = M \sum_{i=1}^{X} \sum_{\text{pa}(x_i)} \sum_{x_i} \hat{p}(x_i, \text{pa}(x_i)) \log \hat{p}(x_i \mid \text{pa}(x_i)) - \frac{\log M}{2} \text{Dim}[g] - DL(g)$$
where \( \hat{p}(x) \) is the percentage of D consistent with \( x \), \( Dim[g] \) is the number of independent parameters in the graph and \( DL(g) \) is the description length of \( g \) (the number of bytes needed to represent \( g \)).

A typical MDL BN learning algorithm is shown in Figure 2. This is the algorithm we use in our experiments. Note that the prior used for MAP is the standard Dirichlet. This adds a little more computation than using MLE, but provides better results. For more information on probabilistic learning, see (Heckerman96).

1. for run ← 0 to max_restarts
   (a) pick a random DAG \( g \)
   (b) parameterize \( g \) using MAP to form \( b \)
   (c) score \( b \) using MDL
   (d) loop
      i. for each DAG \( g' \) differing from \( g \) by adding, removing or reversing an edge, parameterize \( g' \) to form \( b' \), and score \( b' \) using MDL
      ii. pick the \( b' \) with the highest score and replace \( g \) with \( g' \) and \( b \) with \( b' \) if score \( (b') > \) score \( (b) \)
   (e) until no further change \( g \)
2. return the \( b \) with the maximum score among the runs.

**Figure 2: MDL BN learning algorithm**
4. Batch Aggregation Algorithm

We consider expert information to be represented in terms of BNs. Note that it is not far-fetched for different experts to give different BNs; this may be due to differences in individual information sets, different analytical methods, etc. (CW99). So decisions about how to aggregate these BNs to obtain a more general and accurate model are unavoidable.

4.1 Aggregation of joint distributions

The problem of combining joint distributions has long been studied in the statistics and decision analysis community (see (CW99, GZ86) for surveys). Clemen and Winkler (CW99) discuss the key issues involved in combining experts’ probability distributions and review a variety of combination methods from the statistics community. They classify the combination procedures into two types of approaches: behavioral and mathematical. The behavioral approach generates agreement among the experts by having them interact in some way. However, in real life, this approach faces several difficulties: It is difficult to group the experts and have them interact with each other. The manner of the interview may also substantially affect the aggregation result. Finally, the process is time-costly. The mathematical approach uses mathematical methods, such as \textit{LinOP} (Linear Opinion Pool), \textit{LogOP} (Logarithmic Opinion Pool), and Bayesian approaches (CW99), to produce a single joint probability distribution over the individual probability distributions. These aggregation methods only focus on attempting to satisfy abstract properties, such as marginalization and unanimity properties, and the weights introduced are ad hoc, which lead to the lack of semantic support (MC01, PW99).

Maynard-Reid II and Chajewska (MC01) propose a framework in which nature generates samples from a ‘true’ distribution and different experts form their beliefs based on the subsets of the data they have a chance to observe using learning. Hence, the optimal aggregate distribution would be the one learned from the combined sample sets. Such a formulation also leads to a natural way to measure the accuracy of a given aggregation mechanism, that is, to compare the distribution returned by the mechanism with the optimal aggregate distribution.
Let $W$ be a set of possible worlds. For example, if a domain has $n$ binary random variables, then $W$ is the set of $2^n$ possible instantiations of those variables. Each expert or agent is characterized by its distribution over $W$, the number of samples this distribution is based on, and its Dirichlet prior.

**Definition 4:** An agent over $W$ is a tuple $< p, M, \bar{p}, ess >$ where

- $p$ is the agent’s distribution over $W$;
- $M$ is the number of samples the agent is based on;
- $\bar{p}$ is the agent’s prior over $W$; and,
- $ess$ is the agent’s prior estimated sample size.

Let $A$ be the set of all possible agents over $W$. An agent that does not use a prior has $ess = 0$. We refer to $M$ as the size of the agent. A source is an agent that has learned $p$ directly from data.

The sources’ learned distributions are aggregated by the aggregator, also known as the decision maker (DM). Now that each source $i$ has learned a probability model $p_i$, it is given a weight $\alpha_i$ according to the estimated percentage of the data it saw. Intuitively, this corresponds to the relative level of experience of each source. Since only the learned distributions are generally available to the DM and the source data generally is not, we cannot apply the learning algorithm directly. The reason is MLE, MAP, and MDL all depend on the data to be computed – specifically, the empirical distribution over the data. However, Maynard-Reid and Chajewska show that the DM can effectively extract this information from the sources’ distributions.

They adapt the learning algorithms by using a LinOP (Linear Opinion Pool)-based aggregation. LinOP (Stone61) is one of the best-known aggregation operators from statistics. It aggregates a set of distributions by taking their linear weighted sum:

**Definition 5:** Given probability distributions $p_1, \cdots, p_n$ and non-negative parameters $\alpha_1, \cdots, \alpha_n$ such that $\sum \alpha_i = 1$, the LinOP operator is defined such that, for any $w \in W$,

$$\text{LinOP}(\{\alpha_i, p_i\}_{i=1}^n)(w) = \sum_{i=1}^n \alpha_i p_i(w)$$
where $W$ is the set of all joint instantiations over the random variables.

Maynard-Reid II and Chajewska show that $\text{LinOP}$ is the right operator in our framework. Specifically, if each source saw $M_i$ samples and the weights are chosen so that each $\alpha_i = M_i / \sum_i M_i$, i.e., the true percentage of data source $i$ saw, and if both sources and DM use MLE to learn joint distributions, then using $\text{LinOP}$ is essentially equivalent to having the DM learn from the combined data set under some intuitive assumptions: (1) the samples are not noisy and are complete; (2) the individual sample sets are disjoint; (3) the samples for each source are IID – independent and identically distributed; and, (4) the samples in the combined set $D$ are sampled from true distribution $\pi$ and IID. So by applying the $\text{LinOP}$ operator to distributions learned by our sources together with Bayesian inference, we can get the aggregated distribution without the original data.

In the case that the sources and DM use MAP to learn joint distributions and the DM doesn’t trust the source priors the correct aggregation equation is a variation on $\text{LinOP}$:

**Definition 6**: Given probability distributions $p_1, \cdots, p_n$ and $p_D$, prior distribution $\bar{p}_1, \cdots, \bar{p}_n$ and $\bar{p}_D$, estimated samples sizes $\text{ess}_1, \cdots, \text{ess}_n$ and $\text{ess}_D$, non-negative integer $M$, and parameters $\alpha_1, \cdots, \alpha_n$ such that $\sum \alpha_i = 1$, the $\text{PriLin}$ operator is defined such that, for any $w \in W$,

$$ \text{PriLin}(\alpha_i, p_i, \bar{p}_i, \text{ess}_i, \bar{p}_D, \text{ess}_D)(w) = \frac{M}{M + \text{ess}_D} \text{LinOP}(\alpha_i, p_i, \bar{p}_i, \text{ess}_i, \bar{p}_D, \text{ess}_D)(w) + \frac{\text{ess}_i}{M + \text{ess}_D} (p_i(w) - \bar{p}_i(w)) + \frac{\text{ess}_D}{M + \text{ess}_D} \bar{p}_D(w) $$

As in the case of MLE, Maynard-Reid II and Chajewska show that, if the weights are chosen so that each $\alpha_i = M_i / M$, both sources and DM use MAP with the given Dirichlet priors to learn joint distributions, and $M$ is the correct total amount of data seen by the sources, then using $\text{PriLin}$ is equivalent to having the DM learn from the combined data set. Note that $\text{PriLin}$ approaches $\text{LinOP}$ as $M$ gets large. This is not surprising since MAP approaches MLE as $M$ gets large.
4.2 Aggregation of Bayesian networks

Because methods that manipulate joint distributions directly generally do not scale well in practice, especially for large and complex domains, it is more common to first represent distributions in some compact way that facilitates inference, etc. Thus, we assume the use of BNs to graphically represent probabilistic models, reducing the problem of integrating probabilistic models to that of integrating BNs.

Little work has been done in the area of combining BNs. The methods introduced by the statistical community focus on retrieving the full joint distributions from the BNs first, combine the joints, and then learn the BN structure from the combined joint. They are essentially combining joint distributions, and suffer the same problem as discussed in the above section. Most of the work being done in the AI community (MA92, MA93, and PW99) has been built on the ideas related to preserving shared subnetworks. They introduce methods for aggregating BNs in which the aggregator chooses a consensus topology, and then calculates aggregated probabilities. They focus on attempting to satisfy abstract properties such as preserving shared independences and introducing ad hoc parameters, which lead to the conclusion that it is impossible to maintain the consensus structure in the integrated probability model. Therefore these aggregation methods which evaluate a mechanism by how well it satisfies some “intuitive” properties have lacked sufficient semantic underpinnings (MC01, PW99). Many of them also suffer from computational problems, especially in large, complex domains.

Within the above framework, Maynard-Reid II and Chajewska (MC01) also propose a batch aggregation algorithm for combining BNs learned from data. (Note that (RD03) consider a similar setting. Their work differs in that the sources provide priors over structural elements of the DAG which they use to inform the learning process. On the other hand, in Maynard-Reid II and Chajewska’s setting, the sources have distributions learned from data over the domain.) They assume each expert has learned from data a BN using the MDL learning algorithm (see Figure 2). In BN aggregation, they use \( \hat{\text{LinOP}} \) to approximate the \( \hat{\bar{p}} \) used in the parameterization and scoring calculations needed in MDL learning. This is the work we build upon.
Note that a simple way for the parameterization is to retrieve the full joints from all the source BNs, then use LinOP to compute the parameters needed in the aggregate BN. However, this is too expensive, especially for large and complex domains. In their approach, they use the joint-tree inference algorithm, first find which parameters are needed for the aggregate model, applying inference to the source BNs to get the values of these parameters, and then using LinOP to aggregate them. This algorithm can be much more efficient since it is unnecessary to calculate the unneeded parameters. Furthermore, they use caching to store the values of all the previous calculated parameters, so that it isn’t necessary to recalculate the parameters that have been calculated before, making the algorithm even more efficient. As with MAP, $M$ and the Dirichlet priors (in the case of a skeptical DM) are needed, but Maynard-Reid II and Chajewska show experimentally that the final results are not very sensitive to the estimate of $M$. They implemented this BN aggregation algorithm in Matlab. The results of running experiments on some well-known medical diagnosis networks from the literature (the Asia (LS88) and Alarm (BSCC89) networks) showed that the aggregation algorithm consistently improves on the sources and closely approximates the optimal.
5. Iterative Aggregation

One major problem with the batch aggregation algorithm described above is it is not suited to real-world asynchronous environments such as the medical diagnosis domain described in the introduction where we cannot expect information from all sources to arrive simultaneously or even in a short time period. In such environments, the algorithm is slow, inefficient, and imposes time delays. It requires the DM (decision maker) to wait until she has gathered all the information before aggregation can begin, leading to a delayed response to her queries. It would be preferable that aggregation happen in an “anytime” manner, iteratively combining sources as they arrive so that queries can be answered at any time with the response accuracy improving over time.

5.1 Iterative aggregation algorithms

Figure 3 describes a straightforward algorithm that achieves this iterative aggregation by storing all the original sources and re-applying the batch aggregation to all available sources every time new sources arrive.

1. sources ← 0
2. DM ← < ̄p_D, 0, ̄p_D, ess_D >
3. loop indefinitely (or until some termination condition met)
   (a) wait for new sources for some time
   (b) sources ← sources ∪ new sources
   (c) DM ← batch aggregation of sources
4. return DM.

Figure 3: Brute force iterative aggregation algorithm

This algorithm is obviously correct in that the iterative aggregation of a set of sources is independent of the order in which sources arrive with its final result equal to that of batch aggregation. Also, at any time the DM has a defined belief state that can be used to respond to queries. (Note that we initialize the DM’s belief with its prior so that it can begin responding to queries even before sources arrive.) Unfortunately, the algorithm is time-consuming. Batch aggregation is fairly expensive due to its extensive use of probabilistic inference, and the time increases with the number of sources (MC01). For n sources arriving individually, n-1
aggregations will be required, each more expensive than the previous and the last as expensive as the batch aggregation.

To avoid this inefficiency, we modify the existing batch aggregation algorithm to support practical iterative aggregation. The idea is to use the output of the batch aggregation as an intermediate aggregate agent, assigning a new weight to this agent calculated from the original weights of all the information being aggregated. When new information arrives, we combine it with the intermediate agent using the new assigned weight. In this case, we don't have to wait for enough information before aggregation starts, so the DM can consult the current aggregate agent’s model at any time and receive a “current best response” in a timely manner. Also, we don’t have to keep record of all the original information; instead, we only need to keep track of the intermediate aggregate agent, which will save space. When new information arrives, we only need to aggregate the new information with the latest aggregate model, thus avoiding repeatedly aggregating the earlier information it is based on. The resulting algorithm is shown in Figure 4.

Figure 4: Iterative BN aggregation algorithm
5.2 Correctness

We now address the correctness of our algorithm, i.e., whether the result after aggregation equals that of the batch algorithm, independent of the order in which the sources arrive and are aggregated. We show that this is the case if sources and the DM use MLE or MAP to learn joint distributions, and approximately the case if they use MDL to learn a BN.

Let us first consider the case where sources learned joint distributions. We first formally define our (skeptical) MAP aggregation algorithm as a binary operator over the subset of all possible sources $A$.

Definition: Suppose $\overline{\rho}_D$ is a distribution over $W$ and $\text{ess}_D$ is a non-negative real. Then $\bigoplus_{\overline{\rho}_D, \text{ess}_D}$ is the binary operator over $A$ such that for any $S_1 =< p_1, M_1, \overline{\rho}_1, \text{ess}_1 >$ and $S_2 =< p_2, M_2, \overline{\rho}_2, \text{ess}_2 >$ in $A$, $S_1 \bigoplus_{\overline{\rho}_D, \text{ess}_D} S_2 =< p, M, \overline{\rho}, \text{ess} >$ where $M = M_1 + M_2$ and $p = \text{PriLin} \left( \frac{M_i}{M}, p_i, \overline{\rho}_i, \text{ess}_i \right)_{i=1,2} , \overline{\rho}_D, \text{ess}_D \right)$.

Obviously, $A$ is closed under $\bigoplus_{\overline{\rho}_D, \text{ess}_D}$ for all $\overline{\rho}_D$ and $\text{ess}_D$. We now prove that the order in which a set of sources is aggregated does not affect the final result.

Lemma: For any distribution $\overline{\rho}_D$ and non-negative real $\text{ess}_D$, the algebra $< A, \bigoplus_{\overline{\rho}_D, \text{ess}_D} >$ is a commutative semi-group, i.e., satisfies the properties of commutativity and associativity.

Proposition 2: For any distribution $\overline{\rho}_D$ and non-negative real $\text{ess}_D$, the result of applying $\bigoplus_{\overline{\rho}_D, \text{ess}_D}$ iteratively to a set of sources $S \subseteq A$ is invariant under all orderings of $S$ and parenthesizations.

The lemma follows from the fact that the real numbers together with $+$ form a commutative semi-group. The proposition then follows from the Corollary 1.7 in (Hungerford74). That the same is true if sources and DM use MLE follows as a corollary.

Corollary 2.1: Let $A_{\text{MLE}}$ be the subset of $A$ such that if $< p, M, \overline{\rho}, \text{ess } > \in A_{\text{MLE}}$, then $\text{ess } = 0$.

The result of iteratively applying $\bigoplus_{\overline{\rho}_D, 0}$ for any $\overline{\rho}_D$ to a set of $S \subseteq A_{\text{MLE}}$ is invariant under all orderings of $S$ and parenthesizations.

The full proofs of these results can be found in Appendix A.
A couple observations are in order. First, because the binary operators are independent of orderings and parenthesization, the set-based operators are also independent of partitioning, ordering of the subsets, and parenthesizations. That is, a set of sources can arrive in any order of subsets, as long as the subsets partition the set.

Secondly, the independence of parenthesization implies that the result is much stronger than just allowing iterative aggregation. In fact, it allows any groups of sources to be integrated recursively in any combination. That is, any tree where the internal nodes are aggregation operations (corresponding to aggregate agents) and the leaves are groups of sources such that no source appears in more than one leaf will define an expression that gives the same result. In a distributed network setting, this allows us to use a “divide-and-conquer” approach to further lower the total time aggregation takes by creating sub-aggregation agents that operate in parallel. The time savings can be substantial. For example, if each source arrives by itself, our iterative algorithm takes time linear in the number of sources whereas a balanced tree of sub-aggregation agents could take time logarithmic in the number of sources.

Bayesian network MDL learning introduces randomness during learning and aggregation. However, these BNs generally closely approximate distributions learned by MLE or MAP, so these results apply approximately to our algorithm for BNs.

5.3 Properties

Let us consider the major properties of our iterative algorithms.

“Anytime” response: As desired, the iterative aggregation algorithm aggregates whenever new sources arrive and at all times maintains the most up-to-date aggregate model so far. Thus, the DM can respond to queries at any time based on all the information obtained to date.

Practical aggregation: Since we only store and maintain the current aggregate agent and an estimate of the total number of samples the aggregate agent is based on, the DM only needs to combine the aggregate agent with incoming sources when new sources arrive. This sharply reduces the total number of sources involved during each aggregation.
**Idle time utilization:** In an asynchronous environment, we expect there will be some time between the arrival of different (groups of) sources. With the batch algorithm, we can do nothing but remain idle during these wait times. Thus, the actual total time spent is the total wait time plus the high aggregation time. The iterative algorithm, on the other hand, allows us to amortize the aggregation time over this idle time. In the ideal situation, the amount of idle time before the next source arrival equals the amount of time needed to do the previous aggregation.

**Space savings:** The batch algorithm (along with the brute force iterative algorithm) requires that we store all the source BNs at some point. On the other hand, at any point during our iterative aggregation, we only need to store the current aggregate agent and any sources not yet aggregated. Generally speaking, a single aggregate agent BN consumes much less space than the set of source BNs it was created from, so the iterative algorithm can represent huge space savings.

**Accuracy, bias, and the ‘inertia’ effect:** We proved above that if sources and the DM use MLE or MAP, then our algorithm always produces the correct result, independent of arrival order. Unfortunately, this is only approximately true if they use MDL learning to construct BNs. Local optima and the generalization-motivated penalty score in MDL-based aggregation algorithm can introduce errors in early aggregate BNs. If this bias is not corrected quickly, an ‘inertia’ effect can set in such that the increasingly high weight of the aggregate BN makes it difficult for new sources to overcome the bias later on.

We describe in Section 7.1 our experimental exploration of the efficiency and accuracy properties to validate this methodology.
6. BN Aggregation with Structural Prior Knowledge

A second limitation of the batch aggregation algorithm is it does not allow the DM to incorporate into the final aggregate BN any knowledge or partial beliefs she may have about the domain to adapt it to her.

Not only does our algorithm search over the space of BNs which is large, but it relies heavily on BN inference which can still be expensive, especially as the size of the Bayesian network increases. Thus, the original batch aggregation algorithm is inherently very slow. However, in many domains, the aggregator typically has some knowledge in the domain, and wants to represent her partial belief in the final aggregate BN. One solution is to create her model, and combine it with the models from others. However, using our algorithm, this may not satisfy the aggregator, since this requires the specification of a complete BN, and we cannot guarantee that any of this belief can be kept in the aggregate BN. To solve this problem, we extend the batch aggregation algorithm to allow the incorporation of user’s prior knowledge in the form of a partially given structure, which we call PriDAG. We introduce the structural constraints into BN learning during aggregation, which not only makes the algorithm more adaptable, but also improves its performance if the constraints are “correct”.

6.1 PriDAG aggregation algorithm

Our algorithm represents the DM’s prior knowledge as a partially specified BN structure, and keeps this part fixed during the aggregation so that it remains in the aggregate model. This solution will not only satisfy the DM, but also substantially shrink the search space, which will make our algorithm much more efficient. Note that we focus on partial graphical specifications, not allowing any input of pre-set parameters.

To implement this solution, we modified the BN learning algorithm, so that instead of starting with a random DAG, the PriDAG aggregation algorithm starts with a random DAG that satisfies the PriDAG structure. While searching for the best structure, it only searches the candidate DAGs that satisfy the structural constraints from the PriDAG, and discards the other DAGs. The modified BN learning algorithm is shown in Figure 5.
Figure 5: Modified MDL BN learning algorithm that satisfying given structural constraints.

6.2 Properties

The major properties of the PriDAG aggregation algorithm are outlined as follows:

“Semi-automated” aggregation: As desired, the PriDAG aggregation algorithm aggregates all the sources with the incorporation of some structural prior knowledge from the user. Instead of automatically aggregating all the sources, the user is involved in the aggregation process by giving her structural prior knowledge, personalizing the final aggregate model.

Efficient aggregation: In the PriDAG aggregation algorithm, we only need to search those candidate DAGs that satisfy the structural constraints from the PriDAG. This can sharply shrink the search space, making it more efficient than the original batch algorithm when learning the structure of the aggregate model. The degree of efficiency improvement depends on the structure of the PriDAG.

Accuracy: Typically the DM provides her prior knowledge that she knows for certain. We expect that adding true knowledge into the aggregate model will improve the accuracy of the aggregate model, which is based on the belief that prior knowledge of the user will tend to lead to better optima.
In Section 7.2, we describe our experimental exploration of the efficiency and accuracy properties to validate this methodology.
7. Experimental Results

We used the same experimental method described in (MC01) to validate our approach. We simulated sources learning from data by sampling IID source data from an original BN, partitioning it into different data sets, one per source, and then having each source learn a BN from its data set using the BN learning algorithm described above.

We aggregated the learned BNs using both the iterative and batch aggregation algorithms. In the iterative case, we ordered the sources and aggregated them one at a time. We then compared the accuracy of our algorithm with learning from the combined data sets by plotting the Kullback-Leibler (KL) divergence\(^1\) (Kullback59) (also known as KL-distance and relative entropy), a standard measure of “distance” between two distributions, of each distribution from the true distribution. We also compared the time of our iterative algorithm with that of the batch-aggregated BN. We used the well-known Asia network (LS88) for most of our experiments, which is an 8-node BN for a medical diagnosis domain. Preliminary experiments on two other well-known medical diagnosis BNs – the small 5-node Cancer network (Neapolitan90) shown in Figure 3, and the much larger 37-node Alarm network (BSCC89) – produce similar results\(^2\).

We extended the software developed by Maynard-Reid II and Chajewska to incorporate our iterative aggregation algorithm and PriDAG aggregation algorithm. This software is implemented in Matlab 6.5 and relies heavily on Kevin Murphy’s Bayes Net Toolbox\(^3\). We ran our experiments on a Dell computer with a 2.60-GHz Intel® Pentium® 4 CPU, 512MB of RAM, and running Microsoft Windows XP. Our experiments did operate in a networked environment; the Matlab code, experiment input files, and experiment output files were all stored on a separate server. However, since Matlab stores files in memory once they are loaded, and since the I/O during the algorithmic portions of the experiments were minimal, this did not affect our timing results significantly.

\[^1\] The Kullback-Leibler divergence of distribution \(q\) from distribution \(p\) is defined as \(\sum_{w \in \mathcal{W}} p(w) \ln \frac{p(w)}{q(w)}\).

\[^2\] All three are available for free from the Norsys Bayes Net Library at http://www.norsys.com/netlibrary/index.htm.

\[^3\] URL: http://www.ai.mit.edu/~murphyk/Software/BNT/bnt.html
In the following experiments, all sources and the DM used the Dirichlet prior defined by the uniform distribution and an estimated sample size of 1. The DM’s estimates of the sources’ priors and weights and of the total number of samples the sources saw were accurate unless specified otherwise. That is, although in a real application the DM would have to estimate these parameters, it was possible for us to provide the DM with the true values. Also, since significant changes were rare when we used random restarts, we started each structure search with the fully disconnected graph for iterative algorithm and with the given PriDAG for the PriDAG algorithm, and used no random restarts unless specified otherwise.

7.1 Results of iterative aggregation

7.1.1 Comparison of accuracy to sources and batch DM

We first assess how the iterative algorithm compares to the batch algorithm and the sources in accuracy. Ideally, we would like it to perform as well as the batch and improve on all the sources. Figure 6 shows the result of combining the BNs of 15 sources, each learned from 50 samples. We see that the iterative algorithm does as well as all the sources. We also see that its accuracy remains relatively close to that of the batch algorithm. However, we do observe that this closeness decreases as the number of aggregations increases. This is because local optima in the search space and the approximate nature of the MDL score make every aggregation potentially introduce some error into the aggregate. Thus, the multiple aggregations involved in iterative aggregation make the resulting BN tend to diverge more from the true distribution than the batch aggregation BN which is computed directly from the original sources. Figure 7 show the similar result when experimenting on Cancer BN, since all the experiments that we ran on all the three different BNs, i.e., Asia, Cancer and Alarm, have similar results, we omit the results for Cancer and Alarm BN.

Note that in the following figures where the x-axis indicates number of sources for the “iter kl” and “batch kl” data series, it indicates the source number for the “source kl” data series.
To determine the sensitivity of the results to source size, we varied the number of samples that each source saw from 20 to 1000, having sources see the same number of samples. We conducted multiple (three) runs for the settings 20, 50, 100, and 1000, and averaged them. Figure 8 plots the averages for the Asia network.
Figure 8: Comparison of average iterative and batch accuracy over time when iteratively combining same-size sources trained on varied number of samples from the Asia network.

The KL-divergences from both the iterative and batch aggregate BNs to the true distribution become smaller as source size increases. Again, the iterative did a little worse than the batch in each case. However, as one might expect, this difference generally decreases with source size since aggregation introduces fewer errors the more data it is based on.

7.1.2 Aggregate source bias and ‘inertia’

Figures 6, 7 and 8 also illustrate the problem of inertia-maintained bias we discussed in Section 5.3. We see that in each case the incoming sources are unable to change the aggregate agent significantly after a point due to their relatively small weights.

On the other hand, when we varied the number of samples seen by each incoming source between 10 and 2560 samples so that its size equaled the aggregate agent’s and, thus, had the same weight, the result of the iterative aggregation tracked that of the batch very closely. Figure 9 plots the average of experiments on three different randomly generated data sets. When we varied the number of samples between 10 and 4860 so that the incoming source’s
size was twice as large as the aggregate agent, the tracking was even better as we can see in Figure 10. Thus, inertial bias in the aggregate agent can be counteracted if the incoming sources have large weights with respect to the aggregate agent.

Figure 9: Illustration of how inertial bias can be counteracted when sources of weight and size equal to the aggregate agent’s are integrated with it.

Figure 10: Illustration of how inertial bias can be counteracted even more when sources of weight and size twice as large as the aggregate agent’s are integrated with it.
By the same token, if a source with a large weight is added, we would not expect sources with a smaller weight to have much of an effect on the aggregate following the addition. To illustrate this, we constructed experiments in which there is a low-accuracy source whose weight is artificially made ten times that of each of the other sources. Each experiment involved introducing it at a different time relative to the other sources (whose relative position stayed the same), and observing the effect on the sequence of aggregate KL-divergences. Figures 11-13 show the results. In each case, the KL-divergence increases once the large-weight source is added (due to its inaccuracy), and only slowly decreases.

**Figure 11:** Effect on accuracy when a highly weighted, inaccurate source arrives before 10 sources of lower weight and accuracy.
Figure 12: Same as Figure 11, except the highly weighted source arrives midway between the arrivals of the 10 low-weight sources.

![Big-Weight Bad Source at the End](image)

Figure 13: Same as Figure 11, except the highly weighted source arrives after the 10 low-weight sources.

Because we use the estimated percentage of the total number of samples that each source saw as its weight, the relative size of the aggregate agent’s weight keeps increasing. If the weights of the incoming sources are small compared to the aggregate agent, each new aggregate BN will tend to be similar to its predecessor. As we saw above, it is very possible for the aggregate agent’s models to represent local maxima in the search space, which then lead to a biased aggregate result. Combining smaller sources with the aggregate agent increases its weight without changing it significantly, causing an increasingly biased result compared to batch aggregation.

Fortunately, this inertia effect “cuts both ways”; a large-weight, high-accuracy source is also difficult to shake. We repeated the experiment above, but this time used a high-weight source that actually had seen ten times as many samples as each of the other sources, 500 and 50, respectively. Again, we ran the aggregation algorithm multiple times, varying only the position of the large-weight source relative to the other sources. Figure 14 – 16 show the results of the experiments. In each case, the KL-divergence drops significantly once the large-weight source is added (due to its high accuracy), and remains there. The difference between the experiments is much more distinct than in the experiments with the inaccurate source.
because the difference in accuracy between the high- and low-weighted sources is much larger.

Figure 14: Effect on accuracy when a highly weighted, accurate source arrives before 10 sources of lower weight and accuracy.

Figure 15: Same as Figure 14, except the highly weighted source arrives midway between the arrivals of the 10 low-weight sources.
Figure 16: Same as Figure 14, except the highly weighted source arrives after the 10 low-weight sources.

7.1.3 Sensitivity of accuracy to order

The experiments in the previous section imply that our algorithm is sensitive to the order in which sources arrive, at least if their weights vary. Suppose, however, we consider all sources equally informed. Can we expect order-independence under such circumstances? To investigate this, we generated a set of same-size sources, then aggregated them using different orderings. Figures 17 to 19 show the results for three orderings. They show clearly that weight is not the only factor affecting order dependence. In fact, the accuracy of the sources matter as well. As we can see from these figures, sources informed by the same number of samples can vary wildly in their accuracy. Depending on when they arrive, less accurate sources can introduce bias which is then subject to the inertia effect. Of course, the variation in accuracy among same-size sources – and the corresponding order-dependence – tends to decrease as the sizes increase since the source BNs become more accurate.
Figure 17: Variation in aggregate accuracy given random ordering of same-size sources.

Figure 18: Variation in aggregate accuracy given different random ordering of same-size sources.
Figure 19: Variation in aggregate accuracy given increasing KL-divergence order of same-size sources.

Note that there is a tradeoff associated with the ordering of sources: If bad sources arrive early, then undoing any bias they introduce later is easier, but then it is much easier to introduce this bias in the first place. On the other hand, if the low quality sources arrive late, they are less likely to introduce bias into the aggregate, but once introduced, the bias is more difficult to undo later. Efficiently finding the optimal aggregation order for a given set of sources is an open question.

Interestingly, even the batch algorithm is order-dependent. The observant reader may have noticed that, for example, the final batch KL-distance in Figure 11 is slightly higher than the final batch KL-divergence in Figure 13, even though they involve the aggregation of the same sources. This discrepancy is due to the search algorithm visiting pairs of nodes in a specific order determined by a given variable ordering. By default, we pick an ordering based on the topological variable ordering associated with the last source in the vector of sources to integrate. This allows for better comparability with the iterative algorithm. Thus, if the sources differ in their variable orderings, then different orderings of the sources may lead to different parts of the search space. We have found that the variable orderings of less accurate sources tend to lead to worse results. Thus, the algorithm might be improved by using the variable ordering of the source with highest weight.
7.1.4 Time cost

To compare the time cost between iterative and batch aggregation, we aggregated a group of same size sources, varying the number of samples each source saw from 20 to 1000. After each iteration, we compared the cumulative time cost of the iterative algorithm with the time taken by the batch algorithm on the sources seen so far. The result is shown in Figure 20.

![Time Comparison](image)

**Figure 20:** Comparison of total aggregation time of iterative and batch aggregation algorithms as the number of sources increases from 1-15 for fixed source sizes 20, 50, 100, and 1000.

From Figure 20, we can see that the computation time for iterative and batch aggregation are both almost linear in the cumulative size of the agents being aggregated. The slopes of the batch lines are smaller than the iterative lines, implying that fewer sources can be aggregated iteratively during the time taken by batch aggregation. For example, Figure 20 shows that the number of sources integrated iteratively in the time taken by batch aggregation to aggregate 15 varies between 8 and 10 depending on source size (1000 down to 20, respectively). The computation time of both the iterative and batch algorithms increases with the increase of source size, with the time for iterative aggregation growing at a quicker pace than that for batch aggregation. The reason for the higher time requirement and faster time growth in
iterative aggregation is most likely the overhead associated with each new aggregation. For example, iterative aggregation must compute anew with each iteration values that would have been cached when using the batch algorithm. Of course, as we noted earlier, this disadvantage can be substantially offset if there are large wait times between source arrivals.

These experiments explore one extreme of iterative aggregation in which we only aggregate one new source each time. However, in general, multiple sources can be added during each iteration, reducing the number of intermediate aggregations and, thus, potentially saving time. On the other hand, as the number of sources aggregated at each iteration increases, so does the time of each iteration with the extreme case being the time taken when all the sources arrive simultaneously, i.e., the batch algorithm time. Consequently, the iterative algorithm allows for a trade-off between number of iterations and the time per iteration. An interesting open question is what is the optimal grouping of a set of sources to minimize the total aggregation time.

### 7.2 Results of aggregation with PriDAG

To assess the performance of our PriDAG aggregation algorithm, we ran several sets of experiments on the Asia network, varying the number of non-fixed edges. Here, nonfixed edges are the edges on which there is no constraint set in the PriDAG. The PriDAG can be given as a partial BN structure from the DM. In our experiments, it is set by randomly choosing a subset of the true DAG with given number of nonfixed edges. In the PriDAG matrix, \(<i, j> = 0\) means that there cannot be an edge from variable \(i\) to \(j\); \(<i, j> = 1\) means that there must be an edge between \(i\) and \(j\), and the direction is from \(i\) to \(j\); and \(<i, j> = -1\) means that there is no constraint on the edge from \(i\) to \(j\). Since the structure of a BN is a DAG, it is obvious that \(<i, i> must be 0. Thus we set the values in the diagonal of the PriDAG matrix to 0, without counting them in the number of non-fixed edges. Thus, in Asia network, one extreme of the number of non-fixed edges is 0 (the PriDAG has exactly the same structure as the true DAG), and there exists no search space. On the other extreme, the number of nonfixed edges equals 56 (no constraint is set in the PriDAG), and the search space is exactly the same as the one in the original batch algorithm. In this case, the PriDAG aggregation algorithm is equivalent to the original batch algorithm.
Note that we only discarded the elements that are in the diagonal of the PriDAG matrix, not all those that fail the DAG constraint. So the number of nonfixed edges here is only an approximation.

7.2.1 Time saving

To compare the time cost between our PriDAG aggregation algorithm and the original batch algorithm, we aggregated a group of same-size sources, seeing 50 and 1000 samples, and we varied the number of nonfixed edges from 0 to 56. For the same set of sources, we ran the experiments 5 times, and then compared the average number of BNs visited and the average time cost for both batch aggregator and optimal aggregation. The averaged results for the experiments with 50 samples per source are shown in Figure 21 – 23.

![Number of BNs visited vs. number of Nonfixed Edges](image)

**Figure 21: Comparison of number of BNs visited when increasing the number of nonfixed edges in the PriDAG when aggregating 20 sources seeing 50 samples each.**

From Figure 21, we can see that with the increase of the number of non-fixed edges in the PriDAG, i.e., reducing the number of constraints and expanding the searching space, the number of BNs visited when learning the aggregate model increases as well. The bigger the search space, the more time is needed to find the aggregate model structure with the highest MDL score. We expect the time spent on aggregation becomes bigger, which has been testified by our experiment result in Figure 22. Since the time spent on optimal aggregation is
much smaller than that on BN aggregation, to make it clearer, we put the results for optimal aggregation in Figure 23 separately. The experiment aggregating 15 sources seeing 1000 samples each has the same pattern as the one of aggregating 20 sources seeing 50 samples each, so we omit the experiment result here.

Figure 22: Comparison of time spent on aggregations when increasing the number of nonfixed edges in the aggregation of 20 sources seeing 50 samples each.

Figure 23: Comparison of time spent on optimal aggregation when increasing the number of nonfixed edges in the aggregation of 20 sources seeing 50 samples each.
7.2.2 Accuracy comparison

With the same set of experiments described in the above section, we also compared the accuracy of both batch DM and optimal DM when increasing the number of nonfixed edges in the PriDAG and fixing others to match the original BN. The results of both 50-sample source and 1000-sample show that the more true knowledge added into the PriDAG, i.e., the fewer nonfixed edges, the more accurate the aggregate models, although the accuracy improvement is not that much. Figure 24 show the accuracy comparison on the 50-sample sources aggregation when varying the number of nonfixed edges.

![Accuracy Comparison Graph]

**Figure 24:** Comparison of accuracy of aggregate models when increasing the number of nonfixed edges in the aggregation of 20 sources with 50 samples each.
8. Conclusion and Future Work

We have provided iterative extensions of the batch aggregation algorithms developed by Maynard-Reid II and Chajewska (MC01) for aggregating probabilistic models (Bayesian networks in particular) learned from data. Thus, our algorithms support “anytime” querying, utilize idle time, and save space. We proved that the algorithms for joint distributions are independent of the order in which source models arrive. We showed experimentally that the algorithms for aggregating BNs do depend on order and argued that this is due to bias introduced by the greedy optimization in the algorithm and sustained by an “inertial” effect created by the increasing weight given to the aggregate model. However, these experiments also indicate that the accuracy of the resulting models is not significantly reduced, making our algorithm a flexible alternative to the batch version. We have also provided extensions of this batch BN aggregation algorithm to support the incorporation of user’s prior knowledge and save time. We showed experimentally that the more true knowledge is added into the aggregation, the more accurate the aggregate model. Furthermore, the search space becomes smaller and the time spent on aggregation becomes less.

One of the most pressing problems faced by our BN aggregation algorithm is its high computation cost due to its heavy reliance on BN inference. One approach to ameliorating this is to use faster approximate inference algorithms. Another approach involves taking advantage of the general nature of our order-independence results and organizing sources into a hierarchical aggregation tree as described in Section 5.2. Although this does not decrease the overall amount of computation, it allows for much of it to occur in parallel.

Allowing for hierarchical aggregation introduces a second problem: we have assumed that each source appears only once in the aggregation tree, but in some domains, multiple intermediate aggregators may very well hear from the same source. In such a situation, our algorithm essentially “double-counts” each source. A similar problem is that our algorithm does not allow for a source to send an updated model once it has been aggregated into the aggregate model. At the heart of both of these problems is the difficulty in “subtracting” from
an aggregate model the effect of a source. More effective methods than simply storing all sources need to be investigated.

A third problem is a limitation in our PriDAG algorithm: only the structural prior knowledge from the user is allowed. It would be ideal if the aggregation algorithm could incorporate any kind of prior knowledge from the user, qualitative or quantitative.

The fourth related problem is that our algorithms assume sources’ data sets are independent of each other (e.g., no patient sees multiple doctors). This is an assumption we have inherited from the setting in (MC01) and is clearly unrealistic in many domains. Although preliminary experiments indicate that dependence between sources does not necessarily lower the quality of the aggregation algorithms’ results (and sometimes improves it), a principled treatment of dependence remains an open question.
References


Appendix

A. PROOFS

Lemma: For any distribution \( \mathbf{p}_D \) and non-negative real \( \mathbf{ess}_D \), the algebra \(< A, \mathbf{p}_D, \mathbf{ess}_D >\) is a commutative semi-group, i.e., satisfies the properties of commutativity and associativity.

PROOF. Let \( S_1, S_2, S_3 \) be arbitrary elements of \( A \) where \( S_i = < p_i, M_i, \mathbf{p}_i, \mathbf{ess}_i > \). Furthermore, let \( \oplus = \oplus_{\mathbf{p}, \mathbf{ess}} \).

Commutativity: We show that \( S_1 \oplus S_2 = S_2 \oplus S_1 \).
Let \( S_1 \oplus S_2 = < p, \mathbf{p}, \mathbf{ess} > \) and \( S_2 \oplus S_1 = < p', \mathbf{p}', \mathbf{ess}' > \). Then, by the definition of \( \oplus_{\mathbf{p}, \mathbf{ess}} \), \( M = M_1 + M_2 = M_2 + M_1 = M' \), \( \mathbf{p} = \mathbf{p}_D = \mathbf{p}' \), and \( \mathbf{ess} = \mathbf{ess}_D = \mathbf{ess}' \). Furthermore, for each \( w \in W \),
\[
\begin{align*}
p(w) &= \text{PrLin} \left( \left\{ \frac{M_i}{M}, p_i, \mathbf{p}_1, \mathbf{ess}_i \right\}_{i=1,2}, \mathbf{p}_D, \mathbf{ess}_D \right) (w) \\
&= \frac{M}{M + \mathbf{ess}_D} \text{LinOP} \left( \left\{ \frac{M_i}{M}, p_i \right\}_{i=1,2} \right) (w) + \sum_{i=1}^{2} \frac{\mathbf{ess}_i}{M + \mathbf{ess}_D} (p_i(w) - \mathbf{p}_i(w)) + \frac{\mathbf{ess}_D}{M + \mathbf{ess}_D} \mathbf{p}_D(w) \\
&= \frac{M}{M + \mathbf{ess}_D} \sum_{i=1}^{2} \frac{M_i}{M} p_i(w) + \sum_{i=1}^{2} \frac{\mathbf{ess}_i}{M + \mathbf{ess}_D} (p_i(w) - \mathbf{p}_i(w)) + \frac{\mathbf{ess}_D}{M + \mathbf{ess}_D} \mathbf{p}_D(w) \\
&= \frac{M'}{M' + \mathbf{ess}_D} \text{LinOP} \left( \left\{ \frac{M_i}{M'}, p_i \right\}_{i=1,2} \right) p(w) + \sum_{i=1}^{2} \frac{\mathbf{ess}_i}{M' + \mathbf{ess}_D} (p_i(w) - \mathbf{p}_i(w)) + \frac{\mathbf{ess}_D}{M' + \mathbf{ess}_D} \mathbf{p}_D(w) \\
&= \text{PrLin} \left( \left\{ \frac{M_i}{M'}, p_i, \mathbf{p}_1, \mathbf{ess}_i \right\}_{i=2,1}, \mathbf{p}_D, \mathbf{ess}_D, \mathbf{p}_D, \mathbf{ess}_D \right) (w) \\
&= p'(w)
\end{align*}
\]
So \( p = p' \). Thus \( \oplus_{\mathbf{p}, \mathbf{ess}} \) is commutative.

Associativity: We show that \( (S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3) = \mathbf{p}_D, \mathbf{ess} > \).
Let \( (S_1 \oplus S_2) \oplus S_3 = < p, \mathbf{p}, \mathbf{ess} > \)
and \( S_1 \oplus (S_2 \oplus S_3) = < p', \mathbf{p}', \mathbf{ess}' > \).
First, \( M = (M_1 + M_2) + M_3 = M_1 + (M_2 + M_3) = M' \), \( \mathbf{p} = \mathbf{p}_D = \mathbf{p}' \), and \( \mathbf{ess} = \mathbf{ess}_D = \mathbf{ess}' \).
Finally, we show that \( p = p' \). Let \( M_{12} = M_1 + M_2 \), \( \bar{p}_{12} = \bar{p}_D \), \( \text{ess}_{12} = \text{ess}_D \), and

\[
p_{12} = \text{PrLin} \left( \left\{ \frac{M_i}{M_{12}}, p_i, \bar{p}_i, \text{ess}_i \right\}_{i=1,2}, \bar{p}_D, \text{ess}_D \right) \]

Then, for each \( w \in W \),

\[
p(w) = \text{PrLin} \left( \left\{ \frac{M_i}{M}, p_i, \bar{p}_i, \text{ess}_i \right\}_{i=1,2,3}, \bar{p}_D, \text{ess}_D \right)(w)
\]

\[
= \frac{M}{M + \text{ess}_D} \text{LinOP} \left( \left\{ \frac{M_i}{M}, p_i \right\}_{i=1,2,3} \right)(w) + \sum_{i=1,2,3} \frac{\text{ess}_i}{M + \text{ess}_D} \left( p_i(w) - \bar{p}_i(w) \right) + \frac{\text{ess}_3}{M + \text{ess}_D} \bar{p}_D(w)
\]

\[
= \frac{M_{12} + \text{ess}_{12}}{M + \text{ess}_D} p_{12}(w) - \frac{\text{ess}_{12}}{M + \text{ess}_D} \bar{p}_{12}(w) + \frac{M_3 + \text{ess}_3}{M + \text{ess}_D} p_3(w) - \frac{\text{ess}_3}{M + \text{ess}_D} \bar{p}_3(w) + \frac{\text{ess}_D}{M + \text{ess}_D} \bar{p}_D(w)
\]

\[
= \frac{M_{12} + \text{ess}_{12}}{M + \text{ess}_D} \left( \frac{M_{12}}{M_{12} + \text{ess}_D} \text{LinOP} \left( \left\{ \frac{M_i}{M_{12}}, p_i \right\}_{i=1,2} \right)(w) + \sum_{i=1,2} \frac{\text{ess}_i}{M_{12} + \text{ess}_D} \left( p_i(w) - \bar{p}_i(w) \right) + \frac{\text{ess}_3}{M + \text{ess}_D} \bar{p}_D(w) \right)
\]

\[
= \frac{M_{12}}{M + \text{ess}_D} \sum_{i=1}^2 \frac{M_i}{M_{12}} p_i(w) + \sum_{i=1,2} \frac{\text{ess}_i}{M + \text{ess}_D} \left( p_i(w) - \bar{p}_i(w) \right) + \frac{\text{ess}_3}{M + \text{ess}_D} \bar{p}_D(w)
\]

\[
= \sum_{i=1}^3 \left( \frac{M_i + \text{ess}_i}{M + \text{ess}_D} p_i(w) - \frac{\text{ess}_i}{M + \text{ess}_D} \bar{p}_i(w) \right) + \frac{\text{ess}_D}{M + \text{ess}_D} \bar{p}_D(w)
\]

By a similar argument,

\[
p'(w) = \sum_{i=1}^3 \left( \frac{M_i + \text{ess}_i}{M' + \text{ess}_D} p_i(w) - \frac{\text{ess}_i}{M' + \text{ess}_D} \bar{p}_i(w) \right) + \frac{\text{ess}_D}{M' + \text{ess}_D} \bar{p}_D(w)
\]

Since \( M = M' \), \( p(w) = p'(w) \) for all \( w \in W \). Thus \( \oplus \) is associative.

**Proposition 2:** For any distribution \( \bar{p}_D \) and non-negative real \( \text{ess}_D \), the result of applying \( \oplus_{\bar{p}_D, \text{ess}_D} \) iteratively to a set of sources \( S \subseteq A \) is invariant under all orderings of \( S \) and parenthesizations.

**PROOF.** Since \( (A, \oplus_{\bar{p}_D, \text{ess}_D}) \) is a commutative semi-group, the result follows from corollary 1.7 in (Hungerford74).
Corollary 2.1: Let $A_{MLE}$ be the subset of $A$ such that if $<p,M,\bar{p},ess> \in A_{MLE}$, then $ess = 0$.

The result of iteratively applying $\oplus_{\bar{p},0}$ for any $\bar{p}$ to a set of $S \subseteq A_{MLE}$ is invariant under all orderings of $S$ and parenthesizations.

PROOF. $A_{MLE}$ is obviously closed under $\oplus_{\bar{p},0}$ since the estimated sample size parameter of the operator is 0. Furthermore, commutativity and associativity obviously hold since they hold for $(A,\oplus_{\bar{p},0})$ by Lemma 1 and $A_{MLE} \subseteq A$. Thus, $(A_{MLE},\oplus_{\bar{p},0})$ is a commutative semi-group and the result follows from Corollary 1.7 in (Hungerford74).
B. CODE

Our coding is based on the original code of the batch aggregation algorithm using MatLab. We modified some of the files to make the aggregation iterative and allow the incorporation of a user’s structural knowledge in the form of a PriDAG.

For the iterative part, our main experimental goals were to compare the performance of our iterative algorithm with that of the batch algorithm, and to test the effects of sources’ arrival order on the aggregation result, so we added one block of code in the original run_dms.m file for iterative aggregation and also the corresponding code to compare the accuracy of our iterative result with the batch result. Also, we introduced source order parameters in the related files, so that we can use different source orderings for the same set of sources, run several experiments on the same source set, then compare the results to test the effects of ordering in iterative aggregation.

For PriDAG aggregation, we created the files to randomly generate a PriDAG that is a subset of the true DAG given the number of nonfixed edges, to randomly generate a DAG which satisfies the constraints in the PriDAG for the search algorithm to use when using random restart. The corresponding BN learning files were also modified to search only the possible DAGs that satisfy the PriDAG.

In addition to these two big changes, we also modified the original algorithm to make it more efficient and readable, and we fixed some minor bugs in the original code. The main files which were modified and used in our experiments are listed below.

1. params_asia_iter_order_20.00.00.m

```bash
% The parameter file specifies the following:
% general_params structure:
% % (BN or string) input_bn : BN or file specifying BN
% % (BN or string) init_bn : BN or file specifying BN
% string\{num_vars\} var_names : only needed if input_bn is a BN
```
% int num_tot_samples : total number of samples to use
% Boolean rnddag : use random init DAG in learning
% int num_restarts : num restarts in DAG learning
%
% string io : either 'none', 'read', or 'write'; in
% the case of 'read' or 'write', reads/writes total data from/to
% file '<datafile>.dat'; default is 'none'
% string datafile : file to read/write from/to if io
% isn't 'none'
%
% source_params structure:
% ------------------------
% int num_sources
% string distrib_fn : function used to distribute data
% struct{num_settings} constraints : constraints used by distrib_fn
%
% agr_params structure:
% ----------------------
% int{num_settings}[num_sources] num_tot_lagr_samples OR
% string{num_settings} num_tot_lagr_samples : for each setting,
% either 'extract' or the LinOP aggregator's estimate for the
% total number of samples.
% float{num_settings}[num_sources] OR
% string{num_settings} lagr_reliability : for each setting,
% either 'extract' or the reliability weights to use in LinOP
% aggregation.
%
% experiment parameters:
% ----------------------
% int num_runs : num runs per sample setting
% string exp_name : experiment name
% string log_dir : directory for log files
% string out_dir : directory for output files
%
% Author: Pedrito Maynard-Reid II
% Jian Xu
% Date started: 2001 January 7
% Last modified: 2004 August 28

% General parameters.
general_params.input_bn = 'asia';
general_params.var_names = {}; general_params.io = 'write';

% Source parameters.
source_params.sample_num_groups = {...
[50], [50], [50], [50], [50], [50], [50], [50], [50], [50], ...
[50], [50], [50], [50], [50], [50], [50], [50], [50], [50]
};
source_params.num_groups = length(source_params.sample_num_groups);
source_params.rnddag = 0;
source_params.num_restarts = 0;
source_params.order = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20];
% Computing the number of total samples
for i = 1:source_params.num_groups
    source_params.group_total(i) = sum(source_params.sample_num_groups{i});
end;
general_params.num_tot_samples = sum(source_params.group_total);

general_params.datafile = ...
    [general_params.input_bn '.1.2000.dat'];

% Aggregation parameters.
agr_params.rnddag = 0;
agr_params.num_restarts = 0;
agr_params.num_tot_lagr_samples = {'extract'};
agr_params.lagr_reliability = {'extract'};

% Experiment parameters.
num_runs = 1;
exp_name = 'asiaIter.or1.2000.50.00.00.1';
log_dir = "";
out_dir = "";

2. experiments_iter_order.m

function [results, total_time] = experiments_iter_order(param_filename)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% experiments_iter_order(param_filename)
%
% This program compares LinOP-learning BN aggregation with
% sampling-learning aggregation, and compares both against the
% the ideal aggregation of learning from combined original source
% samples. It writes the results to a file in the specified output
% directory. All use MDL to select structure and MAP with uniform
% prior to parameterize the networks.
%
% The parameter file should specify the following:
%
% general_params structure:
% ------------------------
% (BN or string) input_bn : BN or file specifying BN
% (BN or string) init_bn : BN or file specifying BN
% string{num_vars} var_names : only needed if input_bn is a BN
% int num_tot_samples : total number of samples to use
% Boolean rnddag : use random init DAG in learning
% int num_restarts : num restarts in DAG learning
% string io : either 'none', 'read', or 'write'; in
% the case of 'read' or 'write', reads/writes total data from/to
% file '<datafile>.dat'; default is 'none'
% string datafile : file to read/write from/to if io
% isn't 'none'
%
% source_params structure:
% ------------------------
% int num_sources
% string distrib_fn : function used to distribute data
% struct{num_settings} constraints : constraints used by distrib_fn
% int[] order : order of sources' arrival
%
% agr_params structure:
% ----------------------
% int{num_settings}[num_sources] num_tot_lagr_samples OR
% string{num_settings} num_tot_lagr_samples : for each setting,
% either 'extract' or the LinOP aggregator's estimate for the
% total number of samples.
% float{num_settings}[num_sources] OR
% string{num_settings} lagr_reliability : for each setting,
% either 'extract' or the reliability weights to use in LinOP
% aggregation.
%
% experiment parameters:
% ----------------------
% int num_runs : num runs per sample setting
% string exp_name : experiment name
% string log_dir : directory for log files
% string out_dir : directory for output files
%
% Input:
% string param_filename
%
% Output:
% float[num_experiments] results =
% [num_source_samples sources_kl iagr_kl lagr_kl sagr_kl' ...
% iagr_time' lagr_time' sagr_time']
% float total_time : time taken to run experiments
%
% Authors: Pedrito Maynard-Zhang
% Jian Xu
% Date started: 2001 January 2
% Last modified: 2004 September 27
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Load parameters. %%%

eval(param_filename);

rand('state',sum(100*clock));

num_groups = source_params.num_groups;
num_samples_group = source_params.sample_num_groups;

% Setup file I/O.
output_file = [out_dir exp_name ', int2str(num_groups) '.out'];
log_file = [log_dir exp_name ', int2str(num_groups) '.log'];

fid = fopen(output_file, 'w');
delete(log_file);
diary(log_file)

%%% Run experiments. %%%
for i = 1:num_groups
    num_sources = length(num_samples_group{i});
    for j = 1:num_sources
        fprintf(fid, 'num_src_samples(%d %d), ', i, j);
    end;

    for j = 1:num_sources
        fprintf(fid, 'sources_kl(%d %d), ', i, j);
    end;
end;

fprintf(fid, 'iagr_kl, lagr_kl, iagr_time, lagr_time
');

%fprintf(fid, 'lagr_kl, lagr_time
');
total_time = cputime;

% Run aggregators.
[.sources_kl, lagr_kl, lagr_time, tot_lagr_time, iagr_kl, ...
 iagr_time, batch_lagr_kl, batch_lagr_time] = ...
 run_dms_iter_order(general_params, source_params, agr_params)
%results(i,:) = [num_source_samples(i,:) sources_kl(i,:) ...]
 % lagr_kl(i) ...
 % lagr_time(i)];

%%% Write results to file. %%%
for group = 1:num_groups
    num_sources = length(num_samples_group{group});
    for j = 1:num_sources
        fprintf(fid, '%6.0f ', num_samples_group{group}(j));
    end;
    for j = 1:num_sources
        fprintf(fid, '%10.6f ', sources_kl{group}(j));
    end;
end;
fprintf(fid, '%10.6f %10.6f %10.6f %10.2f %10.2f %10.2f %10.2f
', ...
 iagr_kl, lagr_kl(num_groups), batch_lagr_kl, iagr_time, ...
 lagr_time(num_groups), tot_lagr_time, batch_lagr_kl);

sprintf('*****') % log file divider

total_time = cputime - total_time;
fprintf(fid, 'Total time: %f', total_time);

%%% Close files. %%%
diary('off')
fclose(fid);

3. run_dms_iter_order.m

function [sources_kl, lagr_kl, lagr_time, tot_lagr_time, iagr_kl, iagr_time, ...
    batch_lagr_kl, batch_lagr_time] = ...
 run_dms_iter_order(general_params, source_params, agr_params)

% run_dms_iter_order(general_params, source_params, agr_params)
% modified the original run_dms.m: making the aggregation part iterative and changing sources’ order
This function learns sources from samples then creates ideal, Iterative LinOP, Batch LinOP aggregate BNs. It returns the respective KL-distances from the original network and computation times.

Inputs:
% general_params structure:
% -------------------------
% (BN or string) input_bn : BN or file specifying BN
% string{num_vars} var_names : only needed if input_bn is a BN
% int num_tot_samples : total number of samples to use
% string io : either 'none', 'read', or 'write'; in the case of 'read' or 'write', reads/writes total data from/to
% file '<datafile>.dat'; default is 'none'
% string datafile : file to read/write from/to if io isn't 'none'
%
% source_params structure:
% ------------------------
% int num_sources
% string distrib_fn : function used to distribute data
% struct constraints : constraints used by distrib_fn
% Boolean rnddag : use random init DAG in learning
% int num_restarts : num restarts in source DAG learning
% int[] order : order of sources' arrival
%
% agr_params structure:
% ----------------------
% int[num_sources] num_tot_lagr_samples OR
% string num_tot_lagr_samples : either 'extract' or the LinOP aggregator's estimate for the total number of samples.
% float[num_sources] OR
% string lagr_reliability : either 'extract' or the reliability weights to use in LinOP aggregation.
% Boolean rnddag : use random init DAG in learning
% int num_restarts : num restarts in aggregated DAG learning
%
% Outputs:
% float[num_groups][num_sources] sources_kl : KL-dist of sources' BNs
% float iagr_kl : KL-dist of ideal agr BN
% float iagr_time : time to learn ideal aggregate
% float lagr_kl : KL-dist of LinOP agr BN
% float lagr_time : time to aggregate with LinOP
%
% Assumptions:
% o All learners use the same random and num_restarts params.
%
% TO DO:
% o Perhaps make MAP prior/ess variables input parameters.
%
% Authors: Pedrito Maynard-Zhang
% Jian Xu
% Date started: 2001 January 6
% Last modified: 2004 November 16
input_bn = general_params.input_bn;
var_names = general_params.var_names;
num_tot_samples = general_params.num_tot_samples;

rnddag_src = source_params.rnddag;
rnddag_agr = agr_params.rnddag;

num_restarts_src = source_params.num_restarts;
num_restarts_agr = agr_params.num_restarts;

io = general_params.io;
datafile = general_params.datafile;

num_samples_group = source_params.sample_num_groups;
num_groups = source_params.num_groups;
num_tot_lagr_samples = agr_params.num_tot_lagr_samples;
lagr_reliability = agr_params.lagr_reliability;

%%% Setup original network. %%%

sprintf('Setting up original BN...')
if ischar(input_bn)
    % Setup BN from file specification.
    eval(input_bn);
    num_vars = length(var_names);
    [o_bn o_order] = mk_topological_bns(num_vars, var_names, ...
    var_sizes, {connections}, {CPTs});
    orig_bn = o_bn{1};
    orig_order = o_order{1};
    var_sizes = var_sizes(stringmatch(orig_order, var_names));
else
    % input_bn is already a BN.
    num_vars = length(var_names);
    orig_bn = input_bn;
    orig_order = var_names;
    var_sizes = orig_bn.node_sizes;
end;
[orig_num_arcs, orig_num_vs] = dag_properties(orig_bn.dag);

% Print original network.
sprintf('Original')
orig_bn
orig_order
orig_bn.dag
sprintf('%d arcs, %d vs', orig_num_arcs, orig_num_vs)
for i=1:num_vars
    sprintf('Original CPD %s', orig_order{i})
    CPD_to_CPT(orig_bn.CPD{i})
end;

%%% Setup MAP-learning parameters. %%%
% We specify the prior over worlds and estimated sample size for
% computing each source and DMs' Dirichlet parameters.
[source_priors, source_ess, lagr_prior, lagr_ess, ...]
   iagr_prior, iagr_ess] = make_map_vars(source_params);

%%% Generate and distribute source samples. %%%
[source_data, total_data] ...
   = mk_sources_data(orig_bn, num_vars, source_params, general_params);

% Set up aggregation parameters.
if ischar(num_tot_lagr_samples) && strcmp(num_tot_lagr_samples, 'extract')
    % Use true total number of samples in aggregation.
    num_tot_lagr_samples = sum([num_samples_group{:}]);
end; % if
if ischar(lagr_reliability) && strcmp(lagr_reliability, 'extract')
    % Use true reliability parameters in aggregation.
    lagr_reliability = num_samples_group;
else
    alpha = sum([lagr_reliability{:}]);
    for i = 1:num_groups
        lagr_reliability{i} = lagr_reliability{i}/alpha;
    end; % for i
end; % if

% change the coming order of the source groups
order = source_params.order;
source_data = source_data(order);
num_samples_group = num_samples_group(order);

% pumz 2004.10.12 - reorder priors
source_priors = source_priors(order);
source_ess = source_ess(order);
lagr_reliability = lagr_reliability(order)

%%% Iterative linOP
for group = 1:(num_groups)
    %%% Learn sources' BNs from samples. %%%
    num_sources = length(num_samples_group{group});
    for i = 1:(num_sources)
        sprintf('Learning Group %d source %d BN from %d samples...', ...
               group, i, size(source_data{group} {i},1))
        source_times{group}(i) = cputime;
        [source_bns{group} {i} source_orders{group} {i} source_scores{group} {i}] = ... 
        learn_mdl_bn(orig_order, var_sizes, source_data{group} {i}, rnddag_src, ...
               num_restarts_src, source_priors{group} {i}, source_ess{group}(i));
        source_times{group}(i) = cputime - source_times{group}(i);
        sprintf('Time elapsed = %.3f sec', source_times{group}(i))
        [source_num_arcs{group} {i}, source_num_vs{group} {i}] = ...
        dag_properties(source_bns{group} {i}.dag);
    end; % for i

% Print source.
sprintf('Group %d Source %d', group, i)
source_bns{group} {i}
source_orders{group} {i}
source_bns{group} {i}.dag
sprintf('%d arcs, %d vs', source_num_arcs{group} {i}, source_num_vs{group} {i})
for j = 1:num_vars
    sprintf('Group %d Source %d CPD %s', group, i, source_orders{group} {i} {j})

58
CPD_to_CPT(source_bns{group}{i}.CPD{j})
end;

% KL-distances of source from original.
sources_kl{group}(i) = kl_distance(orig_bn, source_bns{group}{i}, ...
        {orig_order, source_orders{group}{i}});
sprintf('Group %d Source %d KL-distance from original = %.3f', group, i, ...
        sources_kl{group}(i))
end;

%%% Apply Iterative LinOP aggregator. %%%

sprintf('Learning Iterative LinOP DM BN Groups %d ...', group)

lagr_time(group) = cputime;
if group == 1
    reliability = [lagr_reliability{1}]
    num_samples = ...
    round(num_tot_lagr_samples*sum(reliability)) %% pumz 2004.11.10
    reliability = mk_stochastic(reliability)
    [lagr_bn{1} lagr_order{1} lagr_score(1)] = linop_agr_bn( ...
        source_bns{1}, source_orders{1}, reliability, ...]
    num_samples, rnddag_agr, num_restarts_agr, ...
        source_priors{1}, source_ess{1}, lagr_prior, lagr_ess);
else
    % - change the position of the aggregated source from the last to the first.
    % %add the aggregated bn as the first sources
    reliability = ...
    [sum([lagr_reliability{1:group-1}]) lagr_reliability{group}];
    num_samples = round(num_tot_lagr_samples*sum(reliability)) %% pumz 2004.11.10
    reliability = mk_stochastic(reliability)
    [lagr_bn{group} lagr_order{group} lagr_score(group)] = ...
        linop_agr_bn([lagr_bn{group-1}] source_bns{group}], ...]
    [lagr_order{group-1}] source_orders{group}], reliability, ...]
    num_samples, rnddag_agr, num_restarts_agr, ...
    [lagr_prior source_priors{group}], ...
    [lagr_ess source_ess{group}], lagr_prior, lagr_ess);
end;
lagr_time(group) = cputime - lagr_time(group);
sprintf('Time elapsed = %.3f sec', lagr_time(group))

[lagr_num_arcs(group), lagr_num_vs(group)] = ...
    dag_properties(lagr_bn{group}.dag);

% Print Iterative LinOP DM.
sprintf('Group %d Iterative LinOP DM', group)
lagr_bn{group}
lagr_order{group}
lagr_bn{group}.dag

for i=1:num_vars
    sprintf('Iterative LinOP DM CPD %s', i)
    CPD_to_CPT(lagr_bn{group}.CPD{i})
end;

% KL-distance of LinOP DM from original.
lagr_kl(group) = kl_distance(orig_bn, lagr_bn{group}, ...
        {orig_order, lagr_order{group}});
sprintf('Iterative LinOP DM KL-distance from original = %.6f, lagr_kl(group))

% KL-distance of Iterative LinOP DM from Last Iteration.
if group > 1
    iter_lagr_kl(group) = kl_distance(lagr_bn{group-1}, lagr_bn{group}, [lagr_order{group-1},
    lagr_order{group}]);
    sprintf('Iterative LinOP DM KL-distance from Last Iteration = %.6f, iter_lagr_kl(group))
end;

% Batch LinOP for the current groups
sprintf('Learning Batch LinOP DM BN for %d Groups ...', group)
reliability = mk_stochastic([lagr_reliability{1:group}])
batch_lagr_time(group) = cputime;
[batch_lagr_bn{group} batch_lagr_order{group} batch_lagr_score(group)] = ...
    linop_agr_bn([source_bns{1:group}], [source_orders{1:group}], ...
    reliability, num_samples, rddag_agr, num_restarts_agr, ...
    [source_priors{1:group}], [source_ess{1:group}], lagr_prior, lagr_ess);
batch_lagr_time(group) = cputime - batch_lagr_time(group);
sprintf('Time elapsed for batch aggragation = %.3f sec', batch_lagr_time(group))
[batch_num_arcs(group), batch_num_vs(group)] = ...
    dag_properties(batch_lagr_bn{group}.dag);

% Print Batch LinOP DM.
sprintf('Batch LinOP DM')
batch_lagr_bn{group}
batch_lagr_order{group}
batch_lagr_bn{group}.dag
sprintf('%d arcs, %d vs', batch_num_arcs(group), batch_num_vs(group))
for i=1:num_vars
    sprintf('Batch LinOP DM CPD %s', batch_lagr_order{group}{i})
    CPD_to_CPT(batch_lagr_bn{group}.CPD{i})
end;

% KL-distance of Batch LinOP DM from original.
batch_lagr_kl(group) = kl_distance(orig_bn, batch_lagr_bn{group}, [orig_order, batch_lagr_order{group}]);
sprintf('Batch LinOP DM KL-distance from original = %.6f, batch_lagr_kl(group))

if group > 1
    bat_prev_kl(group) = kl_distance(batch_lagr_bn{group-1}, batch_lagr_bn{group}, ...
    [batch_lagr_order{group-1}, batch_lagr_order{group}]);
    sprintf('Batch LinOP DM KL-distance from Last Batch = %.6f, bat_prev_kl(group))
end;

% KL-distance of Iterative LinOP DM from Batch LinOP.
iter_batch_kld(group) = kl_distance(batch_lagr_bn{group}, lagr_bn{group}, ...
    [batch_lagr_order{group}, lagr_order{group}]);
sprintf('Iterative LinOP DM KL-distance from batch = %.6f, iter_batch_kld(group))
end;

% Compute the total time for iterative and batch aggragation
tot_lagr_time = sum(lagr_time);
sprintf('Total time elapsed for iterative aggragation = %.3f sec', tot_lagr_time)
tot_batch_time = sum(batch_lagr_time);
sprintf('Total time elapsed for batch aggragation = %.3f sec', tot_batch_time)
%% Learn optimal aggregate network from combined data. 

sprintf('Learning optimal DM BN...
')
iagr_time = cputime;
[iagr_bn iagr_order iagr_score] = learn_mdl_bn(orig_order, var_sizes, ...
   total_data, mddag_agr, num_restarts_agr, iagr_prior, iagr_ess);
iagr_time = cputime - iagr_time;
sprintf('Time elapsed = %.3f sec', iagr_time)
[iagr_num_arcs, iagr_num_vs] = dag_properties(iagr_bn.dag);

% Print optimal DM.
%print('Optimal DM')
iagr_bn
iagr_order
iagr_bn.dag
sprintf('%d arcs, %d vs', iagr_num_arcs, iagr_num_vs) 
for i=1:num_vars
   sprintf('Optimal DM CPD %s', iagr_order{i})
   CPD_to_CPT(iagr_bn.CPD{i})
end;

% KL-distance of optimal DM from original.
iagr_kl = kl_distance(orig_bn, iagr_bn, {orig_order, iagr_order});

% Print out the result
%print('Printing out the result...
')
for group = 1:num_groups
   num_sources = length(num_samples_group{group});
   sprintf('Group %d', group)
   sprintf('Source %d kld = %.6f ;', [1:num_sources; sources_kl{group}])
   sprintf('Source %d num arcs = %d, num vs = %d : ', ...
      [1:num_sources; source_num_arcs{group}; source_num_vs{group}])
   sprintf('Iterative LinOP kld = %.6f, ', lagr_kl(group))
   sprintf('Iterative LinOP time = %.3f, ', lagr_time(group))
   sprintf('Iterative LinOP num arcs = %d, num vs = %d, ', ...
      lagr_num_arcs(group), lagr_num_vs(group))
   sprintf('Batch LinOP kld = %.6f, ', batch_lagr_kl(group))
   sprintf('Batch LinOP time = %.3f, ', batch_lagr_time(group))
   sprintf('Batch LinOP num arcs = %d, num vs = %d, ', ...
      batch_num_arcs(group), batch_num_vs(group))
   sprintf('Iterative LinOP kld from Batch = %.6f, ', iter_batch_kld(group))
   sprintf('Iterative LinOP DM KL-distance from Last Iteration = %.6f, iter_lagr_kl(group))
   sprintf('Batch LinOP DM KL-distance from Last Batch = %.6f, bat_prev_kl(group))
end;

4. mk_sources_data.m

function [source_data, total_data] ...
   = mk_sources_data(orig_bn, num_vars, source_params, general_params)
% mk_sources_data(orig_bn, num_vars, source_params, general_params)
%
% This function assigns the data to different sources
%
% Inputs:
%   general_params structure:
%   -------------------------
%   (BN or string) input_bn    : BN or file specifying BN
%   string{num_vars} var_names : only needed if input_bn is a BN
%   int num_tot_samples        : total number of samples to use
%   Boolean rnddag             : use random init DAG in learning
%   int num_restarts           : num restarts in DAG learning
%   string io                  : either 'none', 'read', or 'write'; in
%   the case of 'read' or 'write', reads/writes total data from/to
%   file '<datafile>.dat'; default is 'none'
%   string datafile            : file to read/write from/to if io
%   isn't 'none'
%
% source_params structure:
% ------------------------
%   int num_sources
%   string distrib_fn        : function used to distribute data
%   struct constraints       : constraints used by distrib_fn
%
% Outputs:
%
% Assumptions:
%   o All learners use the same random and num_restarts params.
%
% TO DO:
%   o Perhaps make MAP prior/ess variables input parameters.
%
% Authors: Pedrito Maynard-Zhang
%                Jian Xu
% Date started: 2004 June 17
% Last modified: 2004 September 13

num_tot_samples = general_params.num_tot_samples;
io = general_params.io;
datafile = general_params.datafile;

num_samples_group = source_params.sample_num_groups;
um_groups = source_params.num_groups;

%%% Generate and distribute source samples. %%%

if strcmp(io, 'read') | strcmp(io, 'write')
    % Open file for I/O.
    if strcmp(io, 'read')
        fid = fopen(datafile, 'r');
    else
        % Close file for I/O.
        fclose(fid);
    end
end
fid = fopen(datafile, 'w');
end;
end;

% Generate data or read it from file, and possibly write
% generated data to file.
if ~strcmp(io, 'read')
    sprintf('Generating %d samples...', num_tot_samples)
    time = cputime;
    rand('state',sum(100*clock));

    % Create all the data samples
    for i = 1 : num_tot_samples
        total_data(i,:) = cell2num(sample_bnet(orig_bn))';
    end;
    elapsed = cputime - time;
    sprintf('Time elapsed for creating data = %.3f sec', elapsed)
    if strcmp(io, 'write')
        % Write data to file.
        sprintf('Writing samples to file...')
        fwrite(fid, total_data');
    end;
else
    % Read data from file.
    sprintf('Reading %d samples from file...', num_tot_samples)
    total_data = fread(fid, [num_vars num_tot_samples]');
    end;
end;

if strcmp(io, 'read') | strcmp(io, 'write')
    % Close file.
    fclose(fid);
end;

% Assign the data to sources
sprintf('Distributing samples...')
    time = cputime;
    num = 0;
    for group = 1:num_groups
        for i = 1 : length(num_samples_group{group})
            temp = num + num_samples_group{group}(i);
            source_data{group}{i} = total_data((num+1):temp, :);
            num = num + num_samples_group{group}(i);
        end;
    end;
    elapsed = cputime - time;
    sprintf('Time elapsed for distributing data = %.3f sec', elapsed)

5. make_map_vars.m

function [source_priors, source_ess, lagr_prior, lagr_ess, ...
    iagr_prior, iagr_ess] = make_map_vars(source_params)
make_map_vars(source_params)

This function setup MAP-learning parameters.

Inputs:

source_params structure:

num_groups
num_samples_group
distrib_fn
constraints

Outputs:

source_priors
source_ess
lagr_prior
lagr_ess

Authors: Pedrito Maynard-Zhang
Jian Xu
Date started: 2004 June 17
Last modified: 2004 October 05

num_samples_group = source_params.sample_num_groups;
num_groups = source_params.num_groups;

var_sizes needed
num_worlds = prod(var_sizes);

Setup MAP-learning parameters.

We specify the prior over worlds and estimated sample size for computing each source and DMs' Dirichlet parameters.

for group = 1 : num_groups
    for i = 1:length(num_samples_group{group})
        source_priors{group}{i} = 'uniform';
        source_ess{group}{i} = 1;
    end;
    source_ess{group} = ones(1, length(num_samples_group{group}));
end;

lagr_prior = 'uniform';
lagr_ess = num_worlds;
lagr_ess = 1;

iagr_prior = 'uniform';
iagr_ess = num_worlds;
iagr_ess = 1;
6. linop_agr_dag_pridag.m

function [max_dag, max_order, max_score, cache] = ...
    linop_agr_dag_pridag(source_bns, var_orders, reliability, num_samples, ... 
    rnddag, num_restarts, source_priors, source_ess, prior, ess, ... 
    cache, pridag, pridag_order)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% linop_agr_dag_pridag(source_bns, var_orders, reliability, 
% num_samples, rnddag, num_restarts, source_priors, 
% source_ess, prior, ess, cache, pridag, pridag_order)
%
% modified the original linop_agr_dag.m: introducing pridag constraints during search
%
% This function learns the aggregate BN DAG using LinOP to approximate
% the MDL score the Decision Maker (DM) would use had she seen all the
% sources' original data. If a Dirichlet prior is provided, it uses
% the corresponding MAP parameters rather than the log-likelihood. It
% uses greedy hill-climbing, starting with an empty network (default)
% or a random network, and doing a specified number of random restarts.
%
% Note: If a cache for the marginals is not provided, it will
% automatically be created.
%
% Inputs:
%   BN{num_sources} source_bns
%   string{num_sources}{num_nodes} var_orders
%   float[num_sources] reliability
%   int num_samples : approximate total number of samples.
%   boolean rnddag : start with random network? (default = false)
%   int num_restarts : number of random restarts  (default = 0)
%   BN/string{num_sources} source_priors : sources' priors over worlds
%       (BN/'uniform'/none';
%       default = all 'none')
%   int[num_sources] source_ess : sources' equivalent sample sizes
%       (default = all num_worlds)
%   BN/string prior : DM prior over worlds (BN/'uniform'/none';
%       default = 'none')
%   int ess : DM equivalent sample sizes (default = num_worlds)
%   MarginalCache{num_sources} cache : (default = empty)
%   PRIDAG pridag : user's prior knowledge in the form of matrix,
%                   where: 1 -- edge, 0 -- no edge, -1 -- don't know
%   string{num_sources} pridag_order : the order of variables in pridag
%
% Output:
%   DAG max_dag
%   string{num_nodes} max_order : cell array of DM's var order
%   float max_score
%   MarginalCache{num_sources} cache
%
% Assumptions:
%   o All sources' BNs are over all variables.
%   o num_samples is finite.
%
% To do:
%   o Add search_depth & num_bn_visited to return list.
num_sources = length(source_bns);
num_vars = length(var_orders{1}); % all BNs have same # vars.
%pumz 2004.07.30
%var_order = var_orders{1}; % arbitrarily start with BN 1's var_order.
%var_sizes = source_bns{1}.node_sizes;
num_worlds = prod(source_bns{1}.node_sizes);

if nargin < 12
    pridag = -1*ones(num_vars, num_vars);  % no special input
    pridag_order = var_orders{1}; % arbitrarily start with BN 1's var_order.
end;

if nargin < 11
    % Create source caches.
    for i = 1:num_sources
        cache{i}{1}.child = [];
    end;
end;

if nargin < 10
    ess = num_worlds;
end;

if nargin < 9
    prior = 'none';
end;

if nargin < 8
    source_ess = ones(1, num_sources)*num_worlds;
end;

if nargin < 7
    for i = 1:num_sources
        source_priors{i} = 'none';
    end;
end;

if nargin < 6
    num_restarts = 0; % default number of random restarts is 0.
end;

if (nargin < 5) | (rnddag == 0)
    var_order = pridag_order;
    % default initial network is only the pridag from user.
    dag = (pridag > 0) + 0;
else
max_fan_in = compute_max_fan_in(var_sizes);
[dag, var_order] = mk_rnd_dag_pridag(pridag, pridag_order, max_fan_in);
% pumz 2004.10.28 - fixed bug of not updated pridag_order
order = stringmatch(var_order, pridag_order);
pridag = pridag(order, order);
pridag_order = var_order;
end;
var_sizes = ... % pumz 2004.10.14
source_bns{1}.node_sizes(stringmatch(var_order, var_orders{1}));

% Score initial DAG.
score = 0;
for i = 1:num_vars
    [params, cache] = compute_lagr_params(source_bns, var_orders, ... 
        reliability, num_samples, var_order, var_sizes, dag, i, ... 
        source_priors, source_ess, prior, ess, cache);
    score = score + mdl_score(i, dag, var_sizes, params, num_samples);
end;

%%% Search. %%%
max_dag = dag;
max_order = var_order;
max_score = score;
num_bn_visited = 1;
search_depth = 1;
for run = 1:num_restarts+1

% Search for higher scoring network.
done = 0;
while ~done
    % Compute successor DAGs, i.e., all those acyclic graphs that 
    % differ from dag by adding, removing, or reversing an 
    % edge. Replace dag with highest higher scoring successor if one 
    % exists; exit otherwise.
    max_sscore = score; % initialize max succ score to node's score.
    for i = 1:num_vars
        for j = 1:num_vars
            if (j ~= i) % ignore self-loops.
                sdag = dag;
                if (sdag(i,j) == 0 && pridag(i,j) == -1)
                    % Check add edge successor.
                    sdag(i,j) = 1;
                    if acyclic(sdag)
                        [params1, cache] = compute_lagr_params(source_bns, ... 
                            var_orders, reliability, num_samples, var_order, ... 
                            var_sizes, dag, j, source_priors, source_ess, ... 
                            prior, ess, cache);
                        [params2, cache] = compute_lagr_params(source_bns, ... 
                            var_orders, reliability, num_samples, var_order, ... 
                            var_sizes, sdag, j, source_priors, source_ess, ... 
                            prior, ess, cache);
                        sscore = score ... 
                            - mdl_score(j, dag, var_sizes, params1, num_samples) ... 
                            + mdl_score(j, sdag, var_sizes, params2, num_samples);
                        num_bn_visited = num_bn_visited + 1;
```

67
```
if sscore > max_sscore
    max_sscore = sscore;
    max_sdag = sdag;
end;
end;

else
    if (sdag(i,j) == 1 && pridag(i,j) == -1)

        % Check delete edge successor.
        sdag(i,j) = 0;
        [params1, cache] = compute_lagr_params(source_bns, ...
            var_orders, reliability, num_samples, var_order, ...
            var_sizes, dag, j, source_priors, source_ess, prior, ...
            ess, cache);
        [params2, cache] = compute_lagr_params(source_bns, ...
            var_orders, reliability, num_samples, var_order, ...
            var_sizes, sdag, j, source_priors, source_ess, prior, ...
            ess, cache);
        sscore = score ...
            - mdl_score(j, dag, var_sizes, params1, num_samples) ...
            + mdl_score(j, sdag, var_sizes, params2, num_samples);
        num_bn_visited = num_bn_visited + 1;
        if sscore > max_sscore
            max_sscore = sscore;
            max_sdag = sdag;
        end;

        % Check reverse edge successor.
        if pridag(j,i) == -1
            sdag(j,i) = 1;
            if acyclic(sdag)
                [params1j, cache] = compute_lagr_params(source_bns, ...
                    var_orders, reliability, num_samples, var_order, ...
                    var_sizes, dag, j, source_priors, source_ess, ...
                    prior, ess, cache);
                [params1i, cache] = compute_lagr_params(source_bns, ...
                    var_orders, reliability, num_samples, var_order, ...
                    var_sizes, dag, i, source_priors, source_ess, ...
                    prior, ess, cache);
                [params2j, cache] = compute_lagr_params(source_bns, ...
                    var_orders, reliability, num_samples, var_order, ...
                    var_sizes, sdag, j, source_priors, source_ess, ...
                    prior, ess, cache);
                [params2i, cache] = compute_lagr_params(source_bns, ...
                    var_orders, reliability, num_samples, var_order, ...
                    var_sizes, sdag, i, source_priors, source_ess, ...
                    prior, ess, cache);
                sscore = score ...
                    - mdl_score(j, dag, var_sizes, params1j, num_samples) ...
                    - mdl_score(i, dag, var_sizes, params1i, num_samples) ...
                    + mdl_score(j, sdag, var_sizes, params2j, num_samples) ...
                    + mdl_score(i, sdag, var_sizes, params2i, num_samples);
                num_bn_visited = num_bn_visited + 1;
                if sscore > max_sscore
                    max_sscore = sscore;
                end;
            end;
        end;
    end;
end;
max_sdag = sdag;
end;
end; % if acyclic(sdag)
end; % if pridag(j,i) == -1
end; % if dag(i,j) = 1
end; % if dag(i,j) = 0
end; % if (j == i)
end; % for j
end; % for i

% Replace dag with highest higher scoring successor if one exists.
if max_sscore > score
  dag = max_sdag;
  score = max_sscore;
  search_depth = search_depth + 1;
else
  done = 1;
end;
end; % while ~done

% Replace max_dag if result of search has higher score.
if score > max_score
  max_dag = dag;
  max_order = var_order;
  max_score = score;
end;

% Create new random network for next restart and score it.
if run < num_restarts + 1
  max_fan_in = compute_max_fan_in(var_sizes);
  [dag, var_order] = mk_rnd_dag_pridag(pridag, pridag_order, max_fan_in);
  var_sizes = ...% pumz 2004.10.28 - fixed bug of not updated pridag_order
  order = stringmatch(var_order, pridag_order);
  pridag = pridag(order, order);
  pridag_order = var_order;
  score = 0;
  for i = 1:num_vars
    [params, cache] = compute_lagr_params(source_bns, var_orders, ...
      reliability, num_samples, var_order, var_sizes, dag, i, ...
      source_priors, source_ess, prior, ess, cache);
    score = score + mdl_score(i, dag, var_sizes, params, num_samples);
  end;
  num_bn_visited = num_bn_visited + 1;
  search_depth = search_depth + 1;
end;
end; % for run
num_bn_visited
search_depth
% Sort DAG topologically.
order = topological_sort(max_dag);
max_dag = max_dag(order, order);
max_order = max_order(order);

7. mk_pridag.m

function [pridag, pridag_order] = ...
    mk_pridag(orig_dag, var_order, num_nonfixed)

num_vars = length(orig_dag);
num_elements = num_vars * num_vars;
pos_nonfixed = 1:(num_elements - num_vars);
pos_nonfixed = pos_nonfixed + floor((pos_nonfixed - 1)/num_vars) + 1; % non-diagonals
pos_nonfixed = pos_nonfixed(randperm(num_elements - num_vars)); % JX random re-order
pos_nonfixed = pos_nonfixed(1:num_nonfixed);
pridag = orig_dag;
pridag(pos_nonfixed) = -1;
pridag_order = var_order;

8. mk_rnd_dag_pridag.m

function [dag, var_order] = ...
    mk_rnd_dag_pridag(pridag, pridag_order, max_fan_in)
% mk_rnd_dag_pridag(pridag, pridag_order, max_fan_in)
%
% This function generates a random BN DAG with some constraints in the
% form of pridag.
%
% Inputs:
%   BN<num_sources> source_bns
%   string<num_sources>+num_nodes> var_orders
%   PRIDAG pridag   : user's prior knowledge in the form of matrix,
%                      where: 1 -- edge, 0 -- no edge, -1 -- don't know
%   string<num_sources> pridag_order    : the order of variables in pridag
%   int max_fan_in  : maximum number of parents each node can have
%                     (default = num_nodes)
%
% Output:
%   DAG dag
%   string<num_nodes> var_order : cell array of DM's var order
%
% Author: Pedrito Maynard-Zhang
%        Jian Xu
% Date started: 2004 August 10
% Last modified: 2004 November 1
num_vars = length(pridag);
if nargin < 3
    max_fan_in = num_vars - 1;
end; % if

% Initialize DAG to have only pridag edges and use pridag order.
dag = (pridag > 0) + 0;
var_order = pridag_order;

% Pick random order. Sort DAG, pridag, and var order to match.
order = randperm(num_vars);
dag = dag(order, order);
pridag = pridag(order, order);
var_order = var_order(order);

% Topologically sort DAG to get final order. Resort pridag and var order
% accordingly.
order = topological_sort(dag);
dag = dag(order, order);
pridag = pridag(order, order);
var_order = var_order(order);

% Pick random arcs, respecting pridag and max_fan_in constraints.
for i = 2:num_vars
    pot_pars = find(pridag(1:i-1, i) == -1); % potential parents
    pot_pars  = pot_pars(randperm(length(pot_pars))); % permute order of potential parents
    max_num_pars = min(max_fan_in, length(pot_pars));
    num_pars = sample_discrete(normalise(ones(1, min(i-1, max_num_pars)+1)))-1;
    dag(pot_pars(1:num_pars),i) = 1; % take first num_pars of (permuted) potential parents
end; % for i

% rnd_matrix = (rand(num_vars) < 0.5) + 0;
function [dag, var_order] = ...
    mk_rnd_dag_pridag_uniform(pridag, pridag_order)

% This function generates a random BN DAG with some constraints in the
% form of pridag.
%
% Inputs:
%   BN{num_sources} source_bns
%   string{num_sources}{num_nodes} var_orders
%   PRIDAG pridag   : user's prior knowledge in the form of matrix,
%   where: 1 -- edge, 0 -- no edge, -1 -- don't know
%   string{num_sources} pridag_order    : the order of variables in pridag
%
% Output:
%   DAG dag
%   string{num_nodes} var_order : cell array of DM's var order
%
% Author: Jian Xu
% Date started: 2004 August 10
% Last modified: 2004 August 10

% num_vars = length(pridag);

var_order = pridag_order;
init_dag = (pridag > 0) + 0;
found = 0;
while ~found
    rnd_matrix = (rand(num_vars) < 0.5) + 0;
    %setting the values of the cells that are -1s in pridag with the
    %values in rnd_matrix
    rnd_dag = ((pridag == -1) + 0).* rnd_matrix;
    %setting values of the cells that are not -1s in pridag.
    dag = init_dag + rnd_dag;
    if acyclic(dag)
        found = 1;
    end;
end;

10. asia.m

% Asia
var_names = {'XRAY', 'BRONCHITIS', 'DYSPNOEA', 'ASIA', 'SMOKER', ...
'LUNG_CANCER', 'TUBERCULOSIS', 'T_C'}
var_sizes = [ 2, 2, 2, 2, 2, 2, 2, 2 ]

XRAY = 1;
BRONCHITIS = 2;
DYSPNOEA = 3;
ASIA = 4;
SMOKER = 5;
LUNG_CANCER = 6;
TUBERCULOSIS = 7;
T_C = 8;
connections = { 'T_C', 'XRAY'; 'SMOKER', 'BRONCHITIS';
  'BRONCHITIS', 'DYSPNOEA'; 'T_C', 'DYSPNOEA';
  'SMOKER', 'LUNG_CANCER'; 'ASIA', 'TUBERCULOSIS';
  'TUBERCULOSIS', 'T_C'; 'LUNG_CANCER', 'T_C';}

CPTs{ASIA} = [ 0.01, 0.99 ];
CPTs{SMOKER} = [ 0.5, 0.5 ];
CPTs{XRAY} = permute( reshape( [0.98, 0.02, 0.05, 0.95], ...
  var_sizes(XRAY), var_sizes(T_C)), [2, 1]);
CPTs{BRONCHITIS} = permute( reshape( [0.6, 0.4, 0.3, 0.7 ], ...
  var_sizes(BRONCHITIS), var_sizes(SMOKER)), [2, 1]);
CPTs{DYSPNOEA} = permute( reshape( [0.9, 0.1, 0.7, 0.3, 0.8, 0.2, 0.1,...
  0.9], var_sizes(DYSPNOEA), var_sizes(BRONCHITIS), var_sizes(T_C)), ...
  [2, 3, 1]);
CPTs{LUNG_CANCER} = permute( reshape( [0.1, 0.9, 0.01, 0.99], ...
  var_sizes(LUNG_CANCER), var_sizes(SMOKER)), [2, 1]);
CPTs{TUBERCULOSIS} = permute( reshape( [0.05, 0.95, 0.01, 0.99], ...
  var_sizes(TUBERCULOSIS), var_sizes(ASIA)), [2, 1]);
CPTs{T_C} = permute( reshape( [1, 0, 1, 0, 1, 0, 0, 1], ...
  var_sizes(T_C), var_sizes(LUNG_CANCER), var_sizes(TUBERCULOSIS)), [2, 3, 1]);

11. cancer.m

% Cancer
% Originally from Cooper84 (PhD thesis), it has appeared in Spiegelhalter86,
% Pearl88 (book, p. 196), and Neapolitan90 (book, p. 179).
% The node names are chosen to match historical choices.
% Neapolitan uses different conditional probabilities; see the
% file 'Cancer_Neapolitan'.

var_names = {'METASTATIC_CANCER', 'SERUM_CALCIUM', 'BRAIN_TUMOR', 'COMA',
  'SEVERE_HEADACHES'}
var_sizes = [ 2, 2, 2, 2, 2 ]
METASTATIC_CANCER = 1;   % in {Present, Absent}
SERUM_CALCIUM = 2;       % in {Increased, Not_Increased}
BRAIN_TUMOR = 3;         % in {Present, Absent}
COMA = 4; % in {Present, Absent}
SEVERE_HEADACHES = 5; % in {Present, Absent}

connections = {'METASTATIC_CANCER', 'SERUM_CALCIUM'; 'METASTATIC_CANCER', 'BRAIN_TUMOR'; 'SERUM_CALCIUM', 'COMA'; 'BRAIN_TUMOR', 'COMA'; 'BRAIN_TUMOR', 'SEVERE_HEADACHES';}

CPTs{METASTATIC_CANCER} = [ 0.2, 0.8 ];

CPTs{SERUM_CALCIUM} = permute( reshape ([ 0.8, 0.2, 0.2, 0.8], var_sizes(SERUM_CALCIUM), ...
var_sizes(METASTATIC_CANCER)), [2, 1]);

CPTs{BRAIN_TUMOR} = permute( reshape ([ 0.2, 0.8, 0.05, 0.95], var_sizes(BRAIN_TUMOR), ...
var_sizes(METASTATIC_CANCER)), [2, 1]);

CPTs{COMA} = permute( reshape ( [0.8, 0.2, 0.8, 0.2, 0.8, 0.2, 0.05, 0.95], var_sizes(COMA), ...
var_sizes(SERUM_CALCIUM), var_sizes(BRAIN_TUMOR)), [2, 3, 1]);

CPTs{SEVERE_HEADACHES} = permute( reshape ( [0.8, 0.2, 0.6, 0.4 ], ...
var_sizes(SEVERE_HEADACHES), var_sizes(BRAIN_TUMOR)), [2, 1]);