

Learning Bayesian Networks from Data: An Efficient Approach Based on Information Theory

Jie Cheng

Dept. of Computing Science
University of Alberta
Alberta, T6G 2H1
Email: jcheng@cs.ualberta.ca

David Bell, Weiru Liu

Faculty of Informatics,
University of Ulster,
UK BT37 0QB
Email: {w.liu, da.bell}@ulst.ac.uk

Abstract

This paper addresses the problem of learning Bayesian network structures from data by using an information theoretic dependency analysis approach. Based on our three-phase construction mechanism, two efficient algorithms have been developed. One of our algorithms deals with a special case where the node ordering is given, the algorithm only require $O(N^2)$ CI tests and is correct given that the underlying model is DAG-Faithful [Spirtes *et. al.*, 1996]. The other algorithm deals with the general case and requires $O(N^4)$ conditional independence (CI) tests. It is correct given that the underlying model is monotone DAG-Faithful (see Section 4.4). A system based on these algorithms has been developed and distributed through the Internet. The empirical results show that our approach is efficient and reliable.

1 Introduction

The Bayesian network is a powerful knowledge representation and reasoning tool under conditions of uncertainty. A Bayesian network is a directed acyclic graph (DAG) with a probability table for each node. The nodes in a Bayesian network represent propositional variables in a domain, and the arcs between nodes represent the dependency relationships among the variables. On constructing Bayesian networks from databases, we use nodes to represent database attributes.

In recent years, many Bayesian network structure learning algorithms have been developed. These algorithms generally fall into two groups, search & scoring based algorithms and dependency analysis based algorithms. An overview of these algorithms is presented in Section 6. Although some of these algorithms can give good results on some benchmark data sets, there are still several problems:

- Node ordering requirement. A lot of previous work assumes that node ordering is available. Unfortunately, in many times this is not the case.
- Lack of efficiency. Some recent algorithms do not need node ordering, but they are generally not very efficient. All practicable dependency analysis based algorithms require exponential numbers of CI tests.
- Lack of publicly available learning tools. Although there are many algorithms for this task, only a few Bayesian network learning systems are publicly available and only one of them (TETRAD II [Spirtes, *et. al.*, 1996]) can be applied to real-world data mining applications where the data sets often have hundreds of variables and millions of records.

This motivates us to do research in the area of Bayesian network structure learning. We developed two algorithms for this task, Algorithm A and Algorithm B. Algorithm A deals with a special case where the node ordering is given, which requires $O(N^2)$ CI tests and is correct given that the underlying model is DAG faithful.

Algorithm B deals with the general case and requires $O(N^4)$ CI tests. It is correct given that the underlying model is monotone DAG faithful. Based on these two algorithms, we have developed a Bayesian network learning system, called Bayesian Network PowerConstructor. The system has been available on the Internet since October 1997 and has already enjoyed over one thousand downloads. Due to the very encouraging experimental results and positive feedback from other users, we plan to expand our work to allow continuous and hidden variables in the underlying models. We also plan to develop a commercial version of our system and integrate it to large data mining, knowledge base and decision support systems.

The remainder of the paper is organized as follows. Section 2 introduces Bayesian network learning from an information theoretic perspective. In Section 3 we present a Bayesian network learning algorithm (Algorithm A) for a special case when node ordering is given. The correctness proof and complexity analysis are also given. Section 4 presents an extension of Algorithm A (called Algorithm B), which does not require node ordering. Then, we give its correctness proof and complexity analysis. In Section 5, we first introduce our Bayesian network learning system (BN PowerConstructor), which implements both of our algorithms. Then, we analyze the experimental results of both algorithms on real-world data sets. Section 6 surveys the algorithms of Bayesian network learning algorithms. Finally, we conclude our work and propose some future research directions in Section 7.

2 Learning Bayesian Networks Using Information Theory

In this section, we first give some basic concepts related with Bayesian network learning. Then we introduce a vital concept used in our approach – d-separation, from an information theoretic perspective.

2.1 Basic Concepts

Definition 2.1. A **directed graph** G can be defined as an ordered pair that consists of a finite set V of nodes and an irreflexive adjacency relation E on V . The graph G is denoted as (V, E) . For each $(x, y) \in E$ we say that there is an arc (directed edge) from node x to node y . In the graph, this is denoted by an arrow from x to y and x and y are called the start point and the end point of the arrow respectively. We also say that node x and node y are **adjacent** or x and y are **neighbors** of each other. x is also called a **parent** of y and y is called a **child** of x . By using the concepts of parent and child recursively, we can also define the concept of **ancestor** and **descendent**. We also call a node that does not have any parent a **root** node. By irreflexive adjacency relation we mean that for any $x \in V$, $(x, x) \notin E$, i.e., an arc cannot have a node as both its start point and end point.

Definition 2.2. In Bayesian network learning, we often need to find a path that connects two nodes without considering the directionality of the edges on the path. To distinguish it from the directed path that connects two nodes by the arcs of a single direction, we call this kind of paths **adjacency paths** or **chains**. This definition is applicable to directed graphs, undirected graphs and mixed graphs.

Definition 2.3. A **directed acyclic graph (DAG)** is a directed graph that contains no directed cycles.

Definition 2.4. For any node in an adjacency path, if two arcs in the path meet at their end point on node v , we call v a **collider** of the path since two arrows ‘collide’ at v . A node that is not a collider of a path is called a **non-collider** of the path. Please note that the concept of collider is always related to a particular path. A node can be a collider in one path and a non-collider in another path.

Definition 2.5. Let $U = \{A, B, \dots\}$ be a finite set of variables with discrete values. Let $P(\cdot)$ be a joint probability function over the variables in U , and let X , Y , and Z be any three subsets of variables in U . X and Y are said to be **conditionally independent** given Z if $P(x | y, z) = P(x | z)$ whenever $P(y, z) > 0$. X and Y are also said to be **independent conditional on Z** .

Definition 2.6. For a probabilistic model M , a graph G is a **dependency map (D-map)** of M if every independence relationship of M can be expressed in G . A graph without any edges is a trivial D-map since every node is independent to all other nodes.

Definition 2.7. For a probabilistic model M , a graph G is an **independency map (I-map)** of M if every independence relationship derived from G is true in M . A complete graph where every node is connected with every other node is a trivial I-map since it contains no independence relationships.

Definition 2.8. A graph G is a **minimum I-map** of a model M if it is an I-map of M and the removal of any arc from G yields a graph that is not an I-map of M .

Definition 2.9. If a graph G is both a D-map and an I-map of M , we call it a **perfect map (P-map)** of M .

Definition 2.10. **Node ordering** is a kind of domain knowledge used by many Bayesian network learning algorithms that specifies a causal or temporal order of the nodes of the graph (variables of the domain), so that any node cannot be a cause or happen earlier than the nodes appearing earlier in the order.

Note that a Bayesian network G of distribution P may not represent all the conditional independence relations of P . When G can actually represent all the conditional independence relations of P , we follow the terminology of [Spirtes, *et. al.*, 1996] to say that P and G are **faithful** to each other. In [Pearl, 1988], G is called a perfect map of P and P is called a **DAG-Isomorph** of G . In [Spirtes *et. al.*, 1996], the authors show that most real-world probabilistic models are faithful to Bayesian networks. In this paper, we are mainly concerned with learning Bayesian networks from data sets that have faithful probabilistic models.

2.2 Learning Bayesian Networks Using Information Theory

As we introduced in Section 1, there are two approaches to Bayesian network learning, search & scoring methods and dependency analysis methods. Our algorithms are based on dependency analysis. Therefore, conditional independence relationships play a very important role in our algorithms. By using the Markov condition, we can get a collection of conditional independence statements from a Bayesian network. These conditional independence statements also imply other conditional independence statements using a group of axioms (see [Pearl, 1998]). By using a concept called **direction dependent separation** or **d-separation** ([Pearl, 1988]), all the valid conditional independence relations can also be directly derived from the topology of a Bayesian network.

Definition 2.11. For a DAG $G = (V, E)$, $X, Y \in V$ and $X \neq Y$, and $C \subset V \setminus \{X, Y\}$, we say that X and Y are **d-separated** given C in G if and only if there exists no adjacency path P between X and Y , such that (i) every collider on P is in C or has a descendent in C and (ii) no other nodes on path P is in C . C is called a cut-set. If X and Y are not d-separated given C we say that X and Y are **d-connected** given C . The definition of d-separation of two nodes can be easily extended to the d-separation of two disjoint sets of nodes. However, because it is not used in our algorithms, we do not give the definition here.

It is proven in [Geiger and Pearl, 1988] that the concept of d-separation can reveal all the conditional independence relations that are encoded in a Bayesian network. In other words, no other criterion can do better.

When learning Bayesian networks from data, we use information theoretic measures to detect conditional independence relations, and then use the concept of d-separation to infer the structures of Bayesian networks.

The definition of d-separation is quite complex, to understand it, we use an intuitive analog from an information theoretic point of view. (A similar analog is suggested in [Spirtes *et. al.*, 1996].)

We view a Bayesian network as a network system of information channels or pipelines, where each node is a **valve** that is either **active** or **inactive** and the valves are connected by noisy **information channels** (edges). The information flow can pass an active valve but not an inactive one. When all the valves on one adjacency path between two nodes are active, we say that this path is **open**. If any valve in the path is inactive, we say that the path is **closed**. Within this framework, we can now explain the concept of d-separation. Since there are two kinds of nodes (colliders and non-colliders) in a Bayesian network, accordingly, the valves in an information flow network have two different initial statuses active and inactive. Initially, any valve that represents a collider of a path is inactive in that path; any valve that represents a non-collider of a path is active in that path. Since a node can be a collider of some paths and non-collider of some other paths, a valve can also be active for some paths and inactive for some other paths. Putting a node in the condition-set can be viewed as altering the status of the corresponding valve and possibly the statuses of other related valves. (Altering the status of a valve can only affect its ancestors that are colliders and are not in the condition-set.) When all the paths between two nodes are closed by altering the statuses of some valves, we say that the two nodes are d-separated by the condition-set corresponding to those valves whose statuses are directly altered.

In our learning algorithms, we measure the volume of the information flow between two nodes to see if a group of valves corresponding to a condition-set can reduce and eventually block the information flow. These results will guide us to construct the correct structure of a Bayesian network from a given data set.

To measure the volume of information flow, we use mutual information and conditional mutual information measurement. In information theory, mutual information is used to represent the expected information gained on sending some symbol and receiving another. In Bayesian networks, if two nodes are dependent, knowing the value of one node will give us some information about the value of the other node. This information gain can be measured using mutual information. Therefore, the mutual information between two nodes can tell us if two nodes are dependent and how close their relationship is. The mutual information of two nodes X_i, X_j is defined as

$$I(X_i, X_j) = \sum_{x_i, x_j} P(x_i, x_j) \log \frac{P(x_i, x_j)}{P(x_i)P(x_j)}, \quad (2.1)$$

and the conditional mutual information is defined as

$$I(X_i, X_j | C) = \sum_{x_i, x_j, c} P(x_i, x_j, c) \log \frac{P(x_i, x_j | c)}{P(x_i | c)P(x_j | c)}, \quad (2.2)$$

where C is a set of nodes. When $I(X_i, X_j)$ is smaller than a certain threshold ϵ , we say that X_i, X_j are marginally independent. When $I(X_i, X_j | C)$ is smaller than ϵ , we say that X_i, X_j are conditionally independent given C .

To learn Bayesian networks from data, our algorithms take as input an ordinary database table. We treat each attribute (field) of the database table as a random variable, which is represented by a node in a Bayesian network. Each record in a database table is a complete instantiation of the random variables in the domain. The marginal and conditional probabilities in Equation (2.1) and Equation (2.2) are estimated using the relative frequencies calculated from the database table.

Assumptions.

Our algorithms make the following assumptions about the database tables that are taken as input.

1. The attributes of a table have discrete values and there are no missing values in all the records.
2. The records occur independently given the underlying probabilistic model of the data.
3. The volume of data is large enough for the reliable CI tests used in our algorithms.

The first assumption states that the variables in a probabilistic model are all discrete random variables and the data set is complete. The second assumption requires that all the records are drawn for a single probabilistic model and are independent of each other. The third assumption guarantees that the statistical decisions made in our algorithms are correct.

3 An Algorithm For Bayesian Network Learning Given Node Ordering

In this section we present Algorithm A, which is used for Bayesian network learning when node ordering is given. Our algorithm for the general case (i.e., when node ordering is not given) will be presented in Section 4.

Algorithm A takes a database table and a complete node ordering as input and constructs a Bayesian network structure as output. The construction process is based on dependence analysis using information theory. A directly related work is the Chow-Liu algorithm (see Section 6), which takes the same input as Algorithm A and constructs a tree structure as output. The Chow-Liu algorithm is also based on information theoretic dependency analysis. In fact, we make use of the Chow-Liu algorithm as the first phase of our algorithm, and then use second and third phases of our own to modify the structure learned using the first phase.

3.1 The Three Phases of Algorithm A

The three phases of this algorithm are drafting, thickening and thinning. In the first phase, this algorithm computes mutual information of each pair of nodes as a measure of closeness, and creates a draft based on this information. In the second phase, the algorithm adds arcs when the pairs of nodes are not conditionally independent on a certain condition-set. The result of Phase II is an I-map of the underlying dependency model given the underlying model is DAG-Faithful. In the third phase, each arc of the I-map is examined using conditional independence tests and will be removed if the two nodes of the arc are conditionally independent. The result of Phase III is a perfect map of the underlying model when the model is DAG-Faithful.

Phase I: (Drafting)

1. Initiate a graph $G(V, E)$ where $V = \{\text{all the attributes of a data set}\}$, $E = \{\}$. Initiate an empty list L .
2. For each pair of nodes (v_i, v_j) where $v_i, v_j \in V$ and $i \neq j$, compute mutual information $I(v_i, v_j)$ using Equation (2.1). For all the pairs of nodes that have mutual information greater than a certain small value ϵ , sort

- them based on their mutual information values and put these pairs of nodes into list L from large to small. Create a pointer p that points to the first pair of nodes in L .
3. Get the first two pairs of nodes of list L and remove them from it. Add the corresponding arcs to E . Move the pointer p to the next pair of nodes. (In this algorithm, the directions of the arcs are decided by the node ordering.)
 4. Get the pair of nodes from L pointed by the pointer p . If there is no open path between the two nodes, add the corresponding arc to E and remove this pair of nodes from L . (A open path is an adjacency path where all the nodes are active. See Section 2.2.)
 5. Move the pointer p to the next pair of nodes and go back to step 4 unless p is pointing to the end of L .

In order to illustrate this algorithm's working mechanism, we use a simple example of a multi-connected network borrowed from [Spirtes *et al.*, 1996]. Suppose we have a database that has underlying Bayesian network as Figure 3.1.a, our task is to rediscover the underlying network from data.

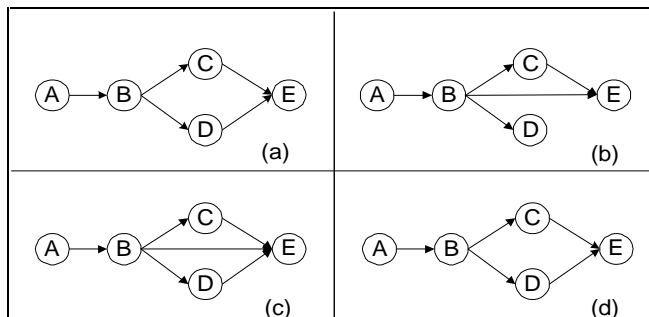


Figure 3.1 A simple multi-connected network and the results after Phase I, II, III of Algorithm A.

After step 2, we can get the mutual information of all 10 pair of nodes. Suppose we have $I(B,D) \geq I(C,E) \geq I(B,E) \geq I(A,B) \geq I(B,C) \geq I(C,D) \geq I(D,E) \geq I(A,D) \geq I(A,E) \geq I(A,C)$, and all the mutual information is greater than ϵ , then list L equals to $\{[B,D], [C,E], [B,E], [A,B], [B,C], [C,D], [D,E], [A,D], [A,E], [A,C]\}$. (In real world situations, the number of pairs of nodes in list L is usually much less than the total number of all possible pairs of nodes, since many pairs of nodes have mutual information smaller than ϵ .) In step 3 to step 5, the algorithm gets a pair of nodes iteratively in the order from L , and it connects the two nodes by an arc and removes the pair of nodes from L if there is no previously existing open path between them. Since every pair of nodes that is connected by an arc is removed from L , at the end of this phase, L equals to $\{[C,D], [D,E], [A,D], [A,E], [A,C]\}$, which are the pairs of nodes that are not directly connected in Phase I but have mutual information greater than ϵ . The draft generated in Phase I is shown in Figure 3.1.b. We can see that the draft is quite close to the real graph. The arc (B, E) is wrongly added and the arc (D, E) is missing.

Our aim of designing this phase is to find a draft that is as close to the real model as possible using only pair-wise mutual information tests, i.e., no conditional independence tests are involved. (The closeness is measured by the number of missing arcs and wrongly added arcs compared to the real model.) Although the draft can be anything from an empty graph to a complete graph without affecting the correctness of the final outcome of the algorithm, a draft that is close to the real underlying model will improve the efficiency of this algorithm.

When searching the draft, we try to minimize the number of missing arcs and the number of wrongly added arcs compared to the real model. Since we only use pair-wise statistics, reducing one kind of errors will often increase the other. We use a stop condition that is a trade-off between the two types of errors – the draft-learning procedure stops when every pair-wise dependency is expressed by an open path in the draft.

The reason that we sort the mutual information from large to small in L is a heuristic, which states that a larger mutual information is more likely representing a direct connection (an arc) than a smaller mutual information, which is more likely representing an indirect connection. In fact, this has been proven correct when the underlying graph is a singly connected graph (a graph without loop). In this case, Phase I of this algorithm is essentially the Chow-Liu algorithm, and it guarantees that the constructed network is the same as the original one, and the second and the third phases will not change anything. Therefore, the Chow-Liu algorithm can be viewed as a special case of our algorithm for the Bayesian networks that have tree structures.

The draft induced in Phase I is the base for Phase II.

Phase II: (Thickening)

6. Move the pointer p to the first pair of nodes in L .
7. Get the pair of nodes ($node1$, $node2$) from L at the position of the pointer p . Call Procedure *find_cut_set* (*current graph*, $node1$, $node2$) to find a cut-set that can d-separate $node1$ and $node2$ in the current graph. Use a conditional independence test to see if $node1$ and $node2$ are conditionally independent given the cut-set. If so, go to next step; otherwise, connect the pair of nodes by adding a corresponding arc to E . (Procedure *find_cut_set* is presented in Section 3.2.)
8. Move the pointer p to the next pair of nodes and go back to step 7 unless p is pointing to the end of L .

In Phase I, we left some pairs of nodes in L only because there are already some open paths between the pairs of nodes. In this phase, we use CI tests and d-separation analysis to see if we should connect those pairs of nodes. Because we only use one CI test to make a decision, we cannot be sure if a connection is really necessary. But we can be sure that no real arcs will be missing after this phase.

In this phase, the algorithm examines all pairs of nodes that remain in L , i.e., the pairs of nodes that have mutual information greater than ϵ and are not directly connected. With every pair of nodes in L , the algorithm first uses Procedure *find_cut_set* to get a cut-set that can d-separate the two nodes, and then uses a CI test to see if the two nodes are independent conditional on the cut-set. Procedure *find_cut_set* tries to find a minimum cut-set (a cut-set with minimum number of nodes) using a heuristic method. This is because that a small condition-set can make the CI test more reliable and more efficient. After the CI test, an arc is added if the two nodes are not conditionally independent. Please note that it is possible that some arcs can be wrongly added in this phase. The reason is that some real arcs may be still missing until the end of this phase and these missing arcs can prevent a proper cut-set from being found. Since the only way for an arc to miss being added is for the nodes to be independent, Phase II will not miss any real arc of the underlying model when the underlying model is DAG-faithful. (The proof of this fact is given in Section 3.3.)

In our example, the graph after Phase II is shown in Figure 3.1.c. Arc (D , E) is added because D and E are not independent conditional on $\{B\}$, which is the smallest cut-set between D and E in the current graph. Arc (A , C) is not added because a CI test reveals that A and C are independent given cut-set $\{B\}$. Arcs (A , D), (C , D) and (A , E) are not added for similar reasons. From Figure 3.1.c we can see that the graph induced after Phase II contains all the real arcs of the underlying model, that is, it is an I-map of the underlying model.

Phase III: (Thinning)

9. For each arc ($node1$, $node2$) in E , if there are other paths besides this arc between the two nodes, remove this arc from E temporarily and call Procedure *find_cut_set* (*current graph*, $node1$, $node2$) to find a cut-set that can d-separate $node1$ and $node2$ in the current graph. Use a conditional independence test to see if $node1$ and $node2$ are conditionally independent given the cut-set. If so, remove the arc permanently; otherwise add this arc back to E .

Since both Phase I and Phase II can add some arcs wrongly, the task of this phase is to identify those wrongly added arcs and remove them. Like in Phase II, we only use one CI test to make a decision. However, this time we can be sure that the decision is correct. The reason is that the current graph is an I-map of the underlying model. This was not true until the end of Phase II. Since we can remove all the wrongly added arcs correctly, the result after Phase III is exactly the same as the underlying model (a perfect map). A proof of this is presented in Section 3.3.

In this phase, the algorithm first tries to find an arc connecting a pair of nodes that are also connected by other paths. (This means that it is possible that the dependency between the pair of nodes could not be due to the direct arc.) Then, the algorithm removes this arc and uses Procedure *find_cut_set* to find a cut-set. After that, a conditional independence test is used to check if the two nodes are independent conditional on the cut-set. If so, remove the arc permanently since the two nodes are independent; otherwise, put the arc back since the cut-set cannot block the information flow between the two nodes. This procedure continues until every such arc is examined.

The ‘thinned’ graph of our example is shown in Figure 3.1.d, which has the structure of the original graph. Edge (B , E) is removed because B and E are independent given $\{C, D\}$. Finally, after three phases, the correct Bayesian network is rediscovered.

3.2 Finding Minimum Cut-Sets

In both Phase II and Phase III, we need Procedure *find_cut_set* (*current graph*, *node1*, *node2*) to get a cut-set between the two nodes, and then use the cut-set as the condition-set in a CI test between *node1* and *node2*. Please recall that a cut-set is always related to the current graph since it is obtained by analyzing the current graph.

Since we use cut-sets as condition-sets in CI tests, minimizing the cardinalities of cut-sets means minimizing the orders of CI tests. From Equation (2.2) we can see that high order CI tests are computationally expensive and can be unreliable when the data set is not large enough. This is the reason why we try to find minimum cut-sets.

Because we know the direction of each arc from the node ordering, we can easily tell if a node is a collider or a non-collider. So, finding a cut-set that can d-separate two nodes is very easy if we do not care about the cardinality of the cut-set. For example, for two nodes *node1*, *node2*, where *node1* appears earlier than *node2* in the order, all the parents of *node2* form a valid cut-set. (See the definition of Markov condition.) In fact, only those parents of *node2* that are on the paths between *node1* and *node2* are enough to block all these paths and form a possibly smaller valid cut-set. However, there can be other smaller cut-sets between *node1* and *node2*. Using the analog of Section 2.2, our task here can be described as blocking the information flow by directly closing as few valves as possible.

In [Acid and Campos, 1996a], a correct algorithm for finding minimum cut-sets is presented, which guarantees that a minimum cut-set between two nodes can always be found. The algorithm first transforms a Bayesian network to an equivalent Markov network and then uses a labeling technique to find the minimum cut-sets. Because their algorithm is quite complex, we develop a heuristic algorithm to do the job, and we now present this procedure.

PROCEDURE *Find_cut_set* (*current graph*, *node1*, *node2*)

Begin

Find all the adjacency paths between *node1* and *node2*;

Store the open paths in open-path-set, store the closed paths in closed-path-set;

Do

While there are open paths that have only one node **Do**

 Store the nodes of each such path in the cut-set;

 Remove all the blocked paths by these nodes from the open-path-set and closed-path-set;

 From the closed-path-set, find paths opened by the nodes in block set and move them to the open-path-set, shorten such paths by removing the nodes that are also in the cut-set;

End While

If there are open paths **Do**

 Find a node that can block the maximum number of the remaining paths and put it in the cut-set;

 Remove all the blocked paths by the node from the open-path-set and the closed-path-set;

 From the closed-path-set, find paths opened by this node and move them to the open-path-set, shorten such paths by removing the nodes that are also in the cut-set;

End If

Until there is no open path

End PROCEDURE.

The above procedure is quite straightforward. It first finds all the adjacency paths between *node1* and *node2* and puts these paths into two sets open-path-set and closed-path-set. Then, it puts the non-colliders that are connected to both *node1* and *node2* in the cut-set since they have to be in every valid cut-set. After that, it uses a heuristic to repeatedly find a node that can block the maximum number of paths and put it in the cut-set, until all the open paths are blocked.

Although this procedure uses a greedy search method, it can find a minimum cut-set in most cases. For instance, on the experiments of ALARM network data, which will be discussed in Chapter 6, all the cut-sets (over 200) found by this procedure are minimal.

In the case when the graph is too complex, e.g., where there are hundreds of paths between a pair of nodes, this procedure can be quite slow since it has to operate on all the paths. In practice, when there are too many paths between a pair of nodes, we replace this procedure by using a cut-set that contains the parents of *node2* that are on the paths between *node1* and *node2*. (Suppose *node2* is not an ancestor of *node1*.) Although the cut-sets found in

this way may be larger than the cut-sets found by Procedure *find_cut_set*, finding the parents of a node takes very little effort and the resulting cut-set is often quite acceptable.

3.3 Correctness Proofs

In addition to the three assumptions made in Section 2.2, we also assume that a probabilistic model actually has a perfect map before we prove that Algorithm A can always find such a perfect map. This assumption is called **DAG-faithfulness** assumption, which is used in the correctness proofs of many other dependency analysis based algorithms. The DAG-faithfulness assumption requires that there exist a DAG that can represent all the dependence and independence relationships in a probabilistic model. (The concept of “faithfulness” is introduced in Section 2.1.)

Suppose a data set that satisfies the three assumptions of Section 2.2 and has a DAG-faithful underlying model M , we can prove proposition 3.1 and proposition 3.2 as follows.

Proposition 3.1 *Graph G_2 generated after Phase II is an I-map of M .*

Proof: Phase I and Phase II of our algorithm examine all the arcs between any two nodes that are not independent, the only way that an arc can be omitted from the graph is that the two nodes for the arc are independent conditional on some condition-set. Hence, any two disconnected nodes in G_2 are conditionally independent in M . Q.E.D.

Proposition 3.2 *Graph G_3 generated after Phase III is a perfect map of M .*

Proof: Because G_2 is an I-map of M , and an arc is removed in Phase III only if the pair of nodes are conditionally independent, it is easy to see that G_3 and all the intermediate graphs of Phase III are I-maps of M . Now, we will prove that G_3 is also a Dependent-map (D-map) of M . Suppose G_3 is not a D-map, then there must exist an arc (a, b) which is in G_3 and for which the two nodes a and b are actually independent in the underlying model M . Therefore, a and b can be d-separated by blocking a set of all the real open paths \mathbf{Pr} in M . In Algorithm A, a and b are connected in G_3 only if a and b are still dependent after blocking a set of all the open paths \mathbf{P} in G_3 . Since all the intermediate graphs of Phase III are I-maps of M , \mathbf{P} includes \mathbf{Pr} and possibly some pseudo-paths. So blocking all the open paths in \mathbf{P} will block all the real open paths in \mathbf{Pr} . Because information cannot pass along the pseudo-paths, the only reason for a and b to be dependent in G_3 is that information can go through the paths in \mathbf{Pr} . This contradicts our assumption that a and b are d-separated by blocking all the open paths of \mathbf{Pr} in M . Thus, G_3 is both a D-map and I-map and hence a perfect map of M . Q.E.D.

The above propositions ensure that our algorithm can construct the *perfect map* of the underlying dependency model, i.e., the induced Bayesian networks are exactly the same as the real underlying probabilistic models of the data sets.

3.4 Complexity Analysis

In this section, we give the time complexity of Algorithm A in term of both the number of CI tests and the number of basic operations. However, the time complexity on the number of basic operations is not a very good index for the comparison of different algorithms since all the algorithms for Bayesian network learning require exponential numbers of basic calculations. Moreover, with the computational power of modern computers, the basic operations like logarithm and multiplication can be executed quickly. In practice, most of the running time of Bayesian network learning algorithms is consumed in data retrieval from databases, which we have found often takes more than 95% of running time. Since data retrieval is done in database engines, there is nothing we can do to improve its efficiency in our algorithms. The only thing we can do is to minimize the number of data retrieving operations. Because each CI test requires one database query and the higher the order of the CI test the slower the query will be, the number of CI tests used and the order of these CI tests are good criteria for judging an algorithm’s performance. In fact, the number of CI tests is a widely used index for comparing different algorithms that are based on dependency analysis.

Since we use mutual information and conditional mutual information tests as CI tests, we will first give the time complexity of such tests on basic operations and then give the time complexity of this algorithm on the number of CI tests. (Mutual information tests can be viewed as CI tests with empty condition-sets.)

Suppose a data set has N attributes, the maximum number of possible values of any attribute is r , and an attribute may have k parents at most. From Equation (2.1) we can get that each computation of mutual information requires $O(r^2)$ basic operations (such as logarithm, multiplication and division) and from Equation (2.2), we can get that

each computation of conditional mutual information requires at most $O(r^{k+2})$ basic operations since a condition-set will have k nodes at most. If a node can have an arbitrary number of parents, the complexity of conditional mutual information test is $O(r^N)$ in the worst case. Clearly, even one CI test can be exponential on the number of basic operations. There are other computational costs in the algorithm, in Phase I, sorting the mutual information of pairs of nodes can be determined in $O(N \log N)$ steps using the *quicksort* algorithm. Procedure *find_cut_set* also requires an exponential number of basic operations for finding the paths between two nodes. However, this procedure can be replaced by a simple $O(N)$ procedure that may return larger cut-sets.

Now we will analyze the time complexity of Algorithm A on the number of CI tests.

Since Phase I computes mutual information between any two nodes, it needs $O(N^2)$ mutual information computations. In Phase II, the algorithm tries to check if it should add arcs to the graph. Each such decision requires one CI test. Therefore, Phase II needs at most $O(N^2)$ number of CI tests. In Phase III, the algorithm tries to see if it can remove the arcs from the graph. Again, a decision requires one CI test and at most $O(N^2)$ number of CI tests is needed. Over all, this algorithm requires $O(N^2)$ CI tests in the worst case.

3.5 Discussion

The Three-Phase Mechanism

Algorithm A is a natural extension of Chow and Liu's tree construction algorithm. It uses the Chow-Liu algorithm as its first phase to create a draft and then uses CI tests in the second and third phases to extend the Chow-Liu algorithm to Bayesian network induction. Under the assumption of DAG-faithfulness, (which is also required by all other dependency analysis based algorithm for their correctness proofs,) Algorithm A guarantees that the perfect maps of the underlying models will be constructed, i.e., the results are exactly the same as the underlying models of the data sets. Moreover, it even preserves the most important merit of the Chow-Liu algorithm – only $O(N^2)$ times of CI tests are needed. In the same situation, all other algorithms except the Wermuth-Lauritzen algorithm, which is highly impracticable, require an exponential number of CI tests. (Some of these algorithms will be introduced in Section 6.) The reason that Algorithm A can avoid exponential complexity is that when checking the dependence relations, it uses only one CI test to make a decision while other algorithms use a group of CI tests. Considering that a correct decision cannot always be made using only one CI test, exponential algorithms are reasonable. However, by using the three-phase mechanism, we managed to use one CI test for a decision all the way until a correct Bayesian network is learned. The method is described as follows. In Phase II, we do not mind that some wrong decisions to be made, though we try to avoid them by generating a draft in Phase I. After Phase II, only one type of error is there, – i.e., the graph we get after Phase II may have wrongly added arcs, but no missing arcs. Given this, as proved in proposition 3.2, Phase III can always make a correct decision using one CI test and finally remove all those arcs added wrong in Phase I and II.

Incorporating Domain Knowledge

A major advantage of Bayesian networks is that the knowledge is represented in a way that agrees with human intuition so that the network structures can be easily understood and the domain knowledge can be easily incorporated. Therefore, a practicable algorithm should always provide a mechanism for incorporating domain knowledge.

Domain knowledge is the experts' knowledge about the structure of a Bayesian network. It can take the following forms:

1. Declare that a node is a root node, i.e., it has no parents.
2. Declare that a node is a leaf node, i.e., it has no children.
3. Declare that a node is a direct cause or direct effect of another node.
4. Declare that a node is not directly connected to another node.
5. Declare that two nodes are independent given a condition-set.
6. Provide partial node ordering, i.e., declare that a node appears earlier than another node in the ordering.
7. Provide a complete node ordering.

Since Algorithm A already uses complete node ordering, partial node ordering becomes unnecessary. Conditional independence information is not very useful either, since this information can be easily obtained using CI tests. Therefore, only the first four types of domain knowledge are important for improving the efficiency and accuracy of the Algorithm A.

As a dependency analysis based algorithm, Algorithm A can be viewed as a constraint based algorithm, which uses CI test results as constraints. Therefore, domain knowledge can be incorporated in a natural way as constraints. For instance, when direct cause and effect relations are available, we can use them as a basis for generating a draft in Phase I. In Phase II, the algorithm will try to add an arc only if it agrees with the domain knowledge. In Phase III, the algorithm will not try to remove an arc if it is specified by domain experts.

DAG-Faithfulness

In Section 3.3, we proved the correctness of this algorithm by using DAG-faithfulness assumption. Although most real-world data sets are actually DAG-faithful [Spirites *et. al.*, 1996], a DAG-unfaithful example can be easily found. For instance [Pearl, 1988], let Z stand for the sound of a bell that rings whenever the outcomes of two fair coins, X and Y , are the same. Using the definition of Bayesian network, $X \rightarrow Z \leftarrow Y$ is the only structure that represents this domain. (Suppose the node ordering is X, Y, Z .) Clearly, the network will not be faithful to any data set of this domain, since the network cannot represent the facts that X and Z and Y and Z are marginally independent. In this case, our algorithm will generate a network that has no arcs at all because that the three nodes are pair-wise independent. Although there are some methods ([Suzuki, 1996]) to solve this problem, these methods are generally inefficient. When a probabilistic model has this kind of sub-structure, our algorithms cannot rediscover this sub-structure correctly. However, Algorithm A can still learn other parts of the network reasonably well.

Conditional Independence Test

In Phase II and Phase III, the algorithm uses CI tests to examine if two nodes are conditionally independent. Since we are only interested in whether two nodes are independent or not, we call this kind of CI tests **qualitative CI tests** so that we can distinguish it from the kind of CI tests we use in Algorithm B below. Although we use conditional mutual information tests as CI tests, other methods for testing independence can do the job equally well. In our experiments, we also tried the Pearson chi-squared test and the likelihood-ratio chi-squared test (see [Agresti, 1990]). The results are very similar to those obtained using conditional mutual information test.

4 An Algorithm For Bayesian Network Learning Without Node Ordering

In this section, we will present Algorithm B, which extends Algorithm A by dropping the requirement of node ordering. This algorithm is also based on our three-phase learning mechanism.

As an extension of Algorithm A, this algorithm takes a database table as input and constructs a Bayesian network structure as output. Since node ordering is not given as input, Algorithm B has to deal with two major problems: (1) how to determine if two nodes are conditionally independent; and (2) how to orient the edges in a learned graph.

For the first problem, there is no easy solution if we want to avoid exponential complexity on CI tests. The difficulty here is that, unlike in Algorithm A, we do not know if a node is a collider or not on a certain path. This prevents us from finding a proper cut-set using a single CI test. To avoid using CI tests blindly on deciding if two nodes are conditionally independent, we developed a method that can distinguish colliders from non-colliders by comparing the results of different CI tests quantitatively, and then determine if two nodes are conditionally independent. In this way, we managed to avoid the exponential complexity on CI tests in Algorithm B. Because we use the results of CI tests quantitatively, we call this kind of test **quantitative CI tests**. Please note that the test itself is always the same, only the way we use it changes.

For the second problem, we developed an edge orientation algorithm that is based on collider identification. Again, to avoid exponential complexity on CI tests for this task, we also have to use the quantitative CI test method. In fact, the procedures for node d-separation and edge orientation in Algorithm B are quite similar since one idea is behind both of them. That is, identifying colliders using quantitative CI tests.

4.1 The Three Phases of Algorithm B

The three phases of this algorithm are the same as those in Algorithm A – they are drafting, thickening and thinning. In the first phase, as for Algorithm A, this algorithm computes mutual information of each pair of nodes as a measure of closeness, and creates a draft based on this information. The draft is a singly connected graph (a graph without loops). In the second phase, the algorithm adds edges to the current graph when the pairs of nodes cannot be separated using a group of CI tests. The result of Phase II contains all the edges of the underlying dependency model given that the underlying model is monotone DAG-faithful, which will be defined in Section 4.4. (Please note that we cannot now talk about I-mapness or P-mapness in this algorithm since we do not know the directions of the edges.) In the third phase, each edge is examined using a group of CI tests and it will be removed if the two nodes of the edge are conditionally independent. The result of Phase III contains exactly the same edges as those in the underlying model when the model is monotone DAG-faithful. At the end of this phase, the algorithm also carries out a procedure to orient the edges of the learned graph. This procedure may not be able to orient all the edges.

Phase I: (Drafting)

1. Initiate a graph $G(V, E)$ where $V = \{\text{all the nodes of a data set}\}$, $E = \{\}$. Initiate an empty list L .
2. For each pair of nodes (v_i, v_j) where $v_i, v_j \in V$ and $i \neq j$, compute mutual information $I(v_i, v_j)$ using Equation (2.1). For all the pairs of nodes that have mutual information greater than a certain small value \mathcal{E} , sort them by their mutual information and put these pairs of nodes into list L from large to small. Create a pointer p that points to the first pair of nodes in L .
3. Get the first two pairs of nodes of list L and remove them from it. Add the corresponding edges to E . Move the pointer p to the next pair of nodes.
4. Get the pair of nodes from L at the position of the pointer p . If there is no adjacency path between the two nodes, add the corresponding edge to E and remove this pair of nodes from L .
5. Move the pointer p to the next pair of nodes and go back to step 4 unless p is pointing to the end of L , or G contains $N - 1$ edges. (N is the number of nodes in G . When G contains $N - 1$ edges, no more edges can be added without forming a loop.)

To illustrate the working mechanism of this algorithm, we use the same example as we use in Section 3.1. Suppose we have a database that has underlying Bayesian network as Figure 4.1.a, our task is to rediscover the underlying network structure from the data. After step 2, we can get the mutual information of all 10 pair of nodes. Suppose (as before) we have $I(B, D) \geq I(C, E) \geq I(B, E) \geq I(A, B) \geq I(B, C) \geq I(C, D) \geq I(D, E) \geq I(A, D) \geq I(A, E) \geq I(A, C)$, and all the mutual information is greater than \mathcal{E} . After step 5, we can construct a draft shown in Figure 4.1.b. Now, list L equals to $[\{B, C\}, \{C, D\}, \{D, E\}, \{A, D\}, \{A, E\}, \{A, C\}]$. In this example, (B, E) is wrongly added and (D, E) and (B, C) are missing because of the existing *adjacency* paths $(D-B-E)$ and $(B-E-C)$. The draft created in this phase is the base for the next phase.

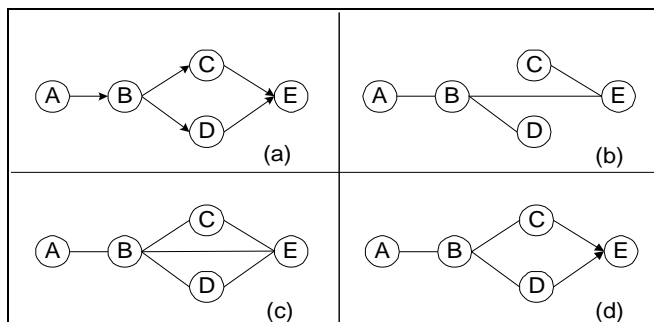


Figure 4.1 A simple multi-connected network and the results after Phase I, II, III of Algorithm B.

As in Algorithm A, the aim of this phase is to find a draft that is close to the underlying model using only pair-wise mutual information tests. The difference is that in this algorithm, we do not try to express every pair-wise dependency by an open path since it is impossible to tell if a path is open or not without knowing the directions of the edges. This phase stops when every pair-wise dependency is expressed by an adjacency path. As a result, there is at most one adjacency path between every pair of nodes. Therefore, the draft is either a polytree or a group of polytrees. (Notice that the draft is different from the draft in Figure 3.1.b, for Algorithm A.) The reason that we sort the mutual information from large to small in L is explained in Section 3.1.

Phase II: (Thickening)

6. Move the pointer p to the first pair of node in L .
7. Get the pair of nodes from L at the position of the pointer p . Call Procedure *try_to_separate_A* (*current graph*, *node1*, *node2*) to see if this pair of nodes can be separated in current graph. If so, go to next step; otherwise, connect the pair of nodes by adding a corresponding edge to E . (Procedure *try_to_separate_A* will be presented in Section 4.2.)
8. Move the pointer p to the next pair of nodes and go back to step 7 unless p is pointing to the end of L .

In Phase I, we left some pairs of nodes in L only because there are adjacency paths between the pairs of nodes. In this phase, we use Procedure *try_to_separate_A* to see whether we should connect those pairs of nodes. As in Algorithm A, we cannot be sure if a connection made is really necessary, but we can be sure that no edges of the underlying model are missing after this phase.

In this phase, the algorithm examines all pairs of nodes that have mutual information greater than \mathcal{E} and are not directly connected. The difference between this algorithm and Algorithm A is that, instead of using one cut-set returned from Procedure *find_cut_set* to check if two nodes are conditionally independent, this algorithm uses a group of CI tests in Procedure *try_to_separate_A* to check that. An edge is *not* added in this phase only when the two nodes can be separated by Procedure *try_to_separate_A*. Please note that two nodes can be separated only when they are independent conditional on a certain cut-set. However, it is possible that some pairs of nodes that cannot be separated are actually conditionally independent. This means that some edges may be wrongly added in this phase. The reasons are as follows.

1. Some real edges may be still missing until the end of this phase, and these missing edges can prevent Procedure *try_to_separate_A* from finding the correct cut-set.
2. Because Procedure *try_to_separate_A* uses a heuristic method, it may not be able to find the correct cut-set for a special group of structures. (The detail will be discussed in Section 4.2.)

In our example, the graph after Phase II is shown in Figure 4.1.c. Edges (B,C) and (D,E) are added because Procedure *try_to_separate_A* cannot separate these pairs of nodes using CI tests. Edge (A,C) is not added because the CI tests can reveal that A and C are independent given cut-set $\{B\}$. Edges (A,D) , (C,D) and (A,E) are not added for the same reason.

Phase III: (Thinning)

9. For each edge in E , if there are other paths besides this edge between the two nodes, remove this edge from E temporarily and call Procedure *try_to_separate_A* (*current graph*, *node1*, *node2*). If the two nodes cannot be separated, add this edge back to E ; otherwise remove the edge permanently.
10. For each edge in E , if there are other paths besides this edge between the two nodes, remove this edge from E temporarily and call Procedure *try_to_separate_B* (*current graph*, *node1*, *node2*). If the two nodes cannot be separated, add this edge back to E ; otherwise remove the edge permanently. (Procedure *try_to_separate_B* will be presented in Section 4.2.)
11. Call Procedure *orient_edges* (*current graph*). (This procedure will be presented in Section 4.3.)

Since both Phase I and Phase II can add edges wrongly, the task of this phase is to identify these wrongly added edges and remove them. In this phase, the algorithm first tries to find each edge connecting a pair of nodes that is also connected by other paths. Since it is possible that the dependency between the pair of nodes is not due to the direct edge, the algorithm removes this edge temporarily and uses Procedure *try_to_separate_A* to check if they can be separated. If so, the edge is removed permanently; otherwise, the edge is restored. This process continues until every such edge is examined. After that, Procedure *try_to_separate_B* is used to double-check all such edges. The reason is that because Procedure *try_to_separate_A* uses heuristic method to find the cut-sets, it may not always be able to separate two conditionally independent nodes. In order to ensure that a correct structure can always be generated, we need to use a correct procedure (Procedure *try_to_separate_B*) at step 10 to re-examine the current edges. Theoretically, we can use Procedure *try_to_separate_B* to replace Procedure *try_to_separate_A* in Phase II and remove step 9 in Phase III since they do the same thing and both of them have complexity $O(N^4)$ on CI tests. (See Section 4.5) But in practice, Procedure *try_to_separate_A* usually uses fewer CI tests and requires smaller condition-sets. Therefore we try to avoid using Procedure *try_to_separate_B* whenever possible.

The ‘thinned’ graph of our example is shown in Figure 4.1.d, which has the same structure as the original graph. Edge (B,E) is removed because B and E are independent given $\{C,D\}$. Given that the underlying dependency model

has a monotone DAG-faithful probability distribution, the structure generated by this procedure contains exactly the same edges as those of the underlying model.

This phase can also orient two out of five edges correctly, i.e., edge (C,E) and edge (D,E) . This result is not very satisfactory. However, due to the limitation of the collider identification based methods, this is the best we can get. That is, no other such method can do better for this structure. Usually, the results will be better if a network structure contains more edges and more colliders. For instance, 42 out of 46 edges can be oriented in the ALARM network (see Section 5.1).

4.2 Separating Pairs of Nodes

In this section, we will present Procedure *try_to_separate_A* and Procedure *try_to_separate_B*, both of which are used to determine if two nodes are conditionally independent. If our procedures can find a proper cut-set that makes the two nodes conditionally independent, we say that we successfully separated the two nodes; otherwise, we say that we failed to separate them. The reason that we fail to separate two nodes can be either that the two nodes are actually connected, or that we cannot find the proper cut-set. Please note that, because we do not know the directions of the edges, we cannot use the term ‘d-separation’.

It is very difficult to find a reasonable cut-set for two nodes in a graph when we do not know the directions of the edges. This problem and the idea behind our solution can be better understood by using the analog we introduced in Section 2.2.

Recall that we can view a Bayesian network as a network of information channels or pipelines. Our algorithms try to build the correct network by first creating a draft and then adding or removing pipelines (edges) based on measuring the information flow. A decision on whether to add or remove a pipeline is made in the following way. To detect if there is a direct pipeline between two nodes, we first try to close all the indirect pipelines between them, and then measure the information flow. If there is still information flow, we assume that there is a direct pipeline between the two nodes in the true model and therefore add a pipeline in the current network. Otherwise, we remove the direct pipeline if it has already been there.

When the node ordering is given, i.e., the directions of edges are known, closing the indirect pipelines is easy. This is because that we know exactly the statuses of the valves in the current network. We can close those indirect pipelines by changing the statuses of some currently active valves to inactive. The only concern here is to change the statuses of as few valves as possible (finding minimum cut-set, Section 3.2).

The situation is quite different when node ordering is not given. This time, we do not know the initial statuses of the valves. As a result, we do not know the outcome of altering the status of a valve either. From the Markov condition we know, if we want to close all the indirect pipelines between two nodes *node1* and *node2*, where *node2* is not an ancestor of *node1*, we can achieve this by making all the neighboring valves of *node2* inactive. Since we do not know the statuses of those valves, one solution is to try every possibility of altering some of them and keeping others unchanged. This is virtually the method used by all other dependency analysis based algorithms, like the SGS algorithm [Spirtes *et. al.*, 1990], the Verma-Pearl algorithm [Verma and Pearl, 1992] and the PC algorithm [Spirtes and Glymour, 1991]. Of course, this requires an exponential number of CI tests.

In our algorithm, we try to avoid the method described above, which uses an exponential number of trials to separate two nodes. But if the result of each trial is only ‘yes’ or ‘no’, there is no way we can avoid the exponential complexity on CI tests. Based on the above consideration, we develop a novel method based on quantitative measurement. The rough idea is like this. We record the amount of information flow given a certain cut-set, which is a subset of all the neighbor valves of *node2* (*node2* is not an ancestor of *node1*). Then, after altering the status of a valve in the neighborhood of *node2*, we measure the information flow again. If the amount is larger, which means that one or more previously closed paths must have been opened, we restore the previous status of this valve and try to alter some other node in the neighborhood of *node2*. If it is smaller, we then can say, “Okay, since the amount of information flow decreased, we must have closed one or more previously open paths. We won’t touch this valve again in the following trials”. By leaving a valve out of further trials permanently, we can avoid to test on all the subsets of the set containing of the nodes in the neighborhood of *node2*. In Section 4.4, we will prove that this quantitative CI test method is correct given the underlying model is monotone DAG-faithful – it is for a specific subset of all DAG-faithful models.

Now we will present the two procedures, Procedure *try_to_separate_A* and Procedure *try_to_separate_B*. The former one is a heuristic procedure, which is efficient but not always correct. The latter one is a correct procedure.

Procedure *try_to_separate_A* (current graph, node1, node2)

1. Find the neighbors of *node1* and *node2* that are on the *adjacency* paths between *node1* and *node2*. Put them into two sets *N1* and *N2* respectively.
2. Remove the currently known child-nodes of *node1* from *N1* and child-nodes of *node2* from *N2*.
3. If the cardinality of *N1* is greater than that of *N2*, swap *N1* and *N2*.
4. Use *N1* as the condition-set *C*.
5. Conduct a CI test using Equation (2.2). Let $v = I(\text{node1}, \text{node2} / C)$. If $v < \epsilon$, return ('separated').
6. If *C* contains only one node, go to step 8; otherwise, for each *i*, let $C_i = C \setminus \{\text{the } i^{\text{th}} \text{ node of } C\}$, $v_i = I(\text{node1}, \text{node2} / C_i)$. Find the smallest value v_m of v_1, v_2, \dots
7. If $v_m < \epsilon$, return ('separated'); otherwise, if $v_m > v$ go to step 8 else let $v = v_m, C = C_m$, go to step 6.
8. If *N2* has not been used, use *N2* as condition-set *C* and go to step 5; otherwise, return ('failed').

From the definition of Bayesian belief networks we know that if two nodes *a*, *b* in the network are not connected, they can be d-separated by the parent nodes of *b* which are on the paths between those two nodes. (We assume that node *b* is not an ancestor of node *a*.) Those parent nodes form a set *P*, which is a subset of *N1* or *N2* of the above procedure. If node ordering is given, we can get *P* immediately and only one CI test is required to check if two nodes are independent. Since this information is not given, we have to use a group of CI tests to find such *P*. By assuming that removing a parent node of *b* from the condition-set will not decrease the mutual information between *a* and *b*, the above procedure tries to find set *P* by identifying and removing the children of *b* and irrelevant nodes from *N1* and *N2* one at a time using a group of computations and comparisons of conditional mutual information. To illustrate the working mechanism of this separating procedure, we use a simple Bayesian network whose true structure is shown in Figure 4.2.

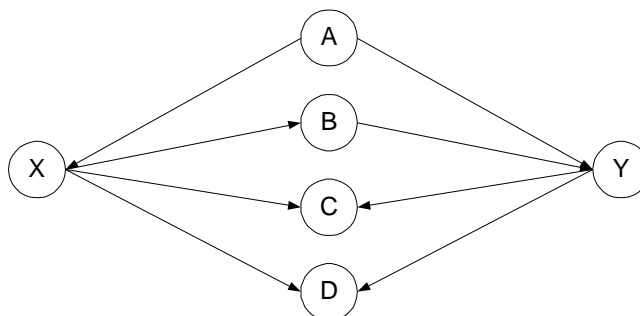


Figure 4.2

Suppose all the edges of the true structure are already discovered and we want to separate *X* and *Y*. If the node ordering is given, i.e., the directions of the edges are known, we can easily see that *A* and *B* are parents of *Y* and *Y* is not an ancestor of *X*. So $P = \{A, B\}$ is a proper cut-set that can separate *X* and *Y*. Since the node ordering is not given in Algorithm B, i.e., the directions of edges are not known, we do not know whether a node is a parent of *Y* or not. In this procedure, we first get *N1* and *N2*, which, happen to be the same in this simple example, are both equal to $\{A, B, C, D\}$. If they are different, we will try to use both of them, and we know at least a subset of one of them should work if the two nodes are not connected, since one of the two nodes to be separated is not an ancestor of the other. In step 5, we use $\{A, B, C, D\}$ as the condition-set *C* and perform a CI test. Since this condition-set closes paths *X-A-Y* and *X-B-Y* but opens *X-C-Y* and *X-D-Y* (see the definition of d-separation in Section 2.2), it cannot separate *A* and *B*. In step 6, we use those subsets of $\{A, B, C, D\}$ that have three nodes as the condition-sets, they are $\{A, B, C\}$, $\{A, B, D\}$, $\{A, C, D\}$, $\{B, C, D\}$. If the data is monotone DAG-faithful (Section 4.4), $\{A, B, C\}$ or $\{A, B, D\}$ will give the smallest value on CI tests. This is because they only leave one path open (path *X-C-Y* and path *X-D-Y* respectively) while other condition-sets leave three paths open. Suppose $\{A, B, C\}$ gives the smallest value, we will never consider node *D* again. In the next iteration, we use those subsets of $\{A, B, C\}$ that have two nodes as the condition-sets, they are $\{A, B\}$, $\{A, C\}$, $\{B, C\}$. After three CI tests, we can finally discover that $\{A, B\}$ can separate *X* and *Y*. Because this procedure can exclude one node out of further trials after each iteration, it can avoid using every subset of *N1* or *N2* as a condition-set, and thus avoid exponential complexity on CI tests.

Since this procedure uses a heuristic method, i.e., it assumes that removing a parent of *b* (where *b* is not an ancestor of *a*) from the condition-set will not decrease the mutual information between *a* and *b*, it may not be able to

separate nodes a and b when the structure satisfies both of the following conditions. (1) There exists at least one path from a to b through a child of b and this child-node is a collider on the path. (2) In such paths, there is one or more colliders besides the child node and all these colliders are the parents or ancestors of b . In such structures, Procedure *try_to_separate_A* may identify a parent of b as a child of b and remove it from P erroneously. As a result, the procedure fails to separate two that can be separated nodes. An example of such a structure is shown in Figure 4.3.

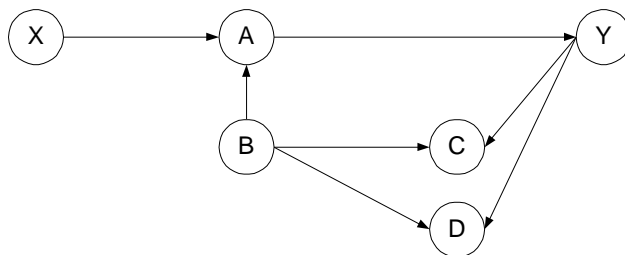


Figure 4.3

Suppose we want to separate X and Y using a certain subset of the neighbor-set of Y , $N2=\{A,C,D\}$. Procedure *try_to_separate_A* will first use $\{A,C,D\}$ as the condition-set, this will leave two paths open, $X-A-B-C-Y$ and $X-A-B-D-Y$. Then, it will use $\{A,C\}$, $\{A,D\}$ and $\{C,D\}$ as the condition-sets. These condition-sets will leave one path open each – it is $X-A-B-C-Y$, $X-A-B-D-Y$ and $X-A-Y$ respectively. If the mutual information between X and Y are the smallest when $X-A-Y$ is open, the procedure will remove node A from further trials. Clearly, this will lead to a failure on separating X and Y . In this example, it happens that the neighbor-set of X , $N1=\{A\}$, can separate X and Y , but there exists some more complex models that this procedure will fail to find using both $N1$ and $N2$. However, such structures are rare in real world situations and this heuristic method works very well in most cases.

To deal with such structures, a correct procedure, Procedure *try_to_separate_B*, is applied in Phase III of the Algorithm B, so that the algorithm ensures that the correct structures can be discovered for all probabilistic models that are monotone DAG-faithful.

Procedure *try_to_separate_B* (current graph, node1, node2)

1. Find the neighbors of *node1* and *node2* that are on the *adjacency* paths between *node1* and *node2*. Put them into two sets $N1$ and $N2$ respectively.
2. Find the neighbors of the nodes in $N1$ that are on the *adjacency* paths between *node1* and *node2*, and do not belong to $N1$. Put them into set $N1'$.
3. Find the neighbors of the nodes in $N2$ that are on the *adjacency* paths between *node1* and *node2*, and do not belong to $N2$. Put them into set $N2'$.
4. If $|N1+N1'| < |N2+N2'|$ let set $C=N1+N1'$ else let $C=N2+N2'$.
5. Conduct a CI test using Equation (2.2). Let $v = I(\text{node1}, \text{node2} / C)$. If $v < \epsilon$, return ('separated') else if C contains only one node return ('failed').
6. Let $C'=C$. For each $i \in [1, |C|]$, let $C_i = C \setminus \{\text{the } i^{\text{th}} \text{ node of } C\}$, $v_i = I(\text{node1}, \text{node2} / C_i)$. If $v_i < \epsilon$ return ('separated') else if $v_i \leq v + e$ then $C' = C' \setminus \{\text{the } i^{\text{th}} \text{ node of } C\}$. (e is a small value.)
7. If $|C'| < |C|$ then let $C=C'$, go to step 5; otherwise, return ('failed').

The major difference between Procedure *try_to_separate_A* and Procedure *try_to_separate_B* is that instead of altering the statuses of the nodes of $N1$ or $N2$ the latter procedure also alters the statuses of the nodes of set $N1'$ or $N2'$. Since altering the statuses of two consecutive nodes in a path can always close the path (two consecutive nodes cannot both be colliders in a path), blocking set $N1+N1'$ or $N2+N2'$ can close all the paths that connect *node1* and *node2* through two or more nodes. The only open paths are those connect *node1* and *node2* through one collider. Under this circumstance, we can remove all the colliders connecting *node1* and *node2* without opening any previously closed paths. Thus, all paths between *node1* and *node2* in the underlying model can be closed. For the example shown in Figure 4.3, this procedure will first use $\{A,B,C,D\}$ as the condition-set. Clearly, X and Y can be successfully separated using this condition-set. The correctness proof of this procedure is in Section 4.4.

In both Procedure *try_to_separate_A* and Procedure *try_to_separate_B*, we have to compare the mutual information on different condition-sets. If we do not use quantitative CI tests, we cannot compare the results of different CI tests and therefore cannot remove irrelevant nodes. By reducing the cardinalities of the condition-sets after each iteration, we can avoid testing on every subset of the initial condition-set and thus avoid the exponential

complexity on CI tests. On the other hand, qualitative CI test based algorithms have to carry out the test on every subset of $N1$ or $N2$ in order to separate two nodes, so the number of CI tests in such algorithms must be exponential in the worst case.

4.3 Orienting Edges

Among the nodes in Bayesian networks, only colliders can let information flow pass them when they are instantiated. The working mechanism of collider identification based methods is described as follows. For any three nodes X , Y and Z of a Bayesian network, if X and Y and Y and Z are directly connected and X and Z are not directly connected, the structure formed by X , Y and Z is called the **V-structure**. There are only three possible types of V-structures, (1) $X \rightarrow Y \rightarrow Z$, (2) $X \leftarrow Y \rightarrow Z$ and (3) $X \rightarrow Y \leftarrow Z$. Among them, only the third type can let information pass from X to Z when Y is instantiated. In other words, only the third type makes X and Z dependent conditional on $\{Y\}$.

Using this characteristic of Bayesian networks, we can identify all the V-structures of the third type in a network and orient the edges in such structures using CI tests. Then, we can try to orient as many edges as possible using these identified colliders. Clearly, the collider identification based methods may not be able to orient all the edges in a network. The number of edges can be oriented is limited by the structure of the network. In an extreme case, when the network does not contain any V-structure of the third type, these methods cannot orient any edges at all. However, this method is quite popular among Bayesian network learning algorithms due to its efficiency and reliability. There are a few algorithms [Lam and Bacchus, 1994; Friedman and Goldszmidt, 1996] that use pure search & scoring methods to search for the correct directions of the edges. But these methods are generally slower than collider identification based methods since the search space is much larger when node ordering is not given.

The collider identification based methods have been used in many algorithms, e.g., the Rebane-Pearl algorithm [Rebane and Pearl, 1987], the SGS algorithm [Spirtes *et. al.*, 1990], the Verma-Pearl algorithm [Verma and Pearl, 1992] and the PC algorithm [Spirtes and Glymour, 1991]. Apart from the method used in the Rebane-Pearl algorithm, which is used for edge orientation in polytrees, all the edge orientation methods for general Bayesian networks have exponential complexity on CI tests. To avoid exponential complexity, we developed a quantitative CI test based method for collider identification. The idea behind this method is the same as that in Procedure *try_to_separate_B* – i.e., by closing all the paths between two nodes that have length equal to or greater than three, we can always identify the colliders between them in a certain order.

Procedure *orient_edges* (*current graph*)

1. For any two nodes $s1$ and $s2$ that are not directly connected and where there is at least one node that is the neighbor of both $s1$ and $s2$, find the neighbors of $s1$ and $s2$ that are on the *adjacency* paths between $s1$ and $s2$. Put them into two sets $N1$ and $N2$ respectively.
2. Find the neighbors of the nodes in $N1$ that are on the *adjacency* paths between $s1$ and $s2$, and do not belong to $N1$. Put them into set $N1'$.
3. Find the neighbors of the nodes in $N2$ that are on the *adjacency* paths between $s1$ and $s2$, and do not belong to $N2$. Put them into set $N2'$.
4. If $|N1+N1'| < |N2+N2'|$ let set $C=N1+N1'$ else let $C=N2+N2'$.
5. Conduct a CI test using Equation (2.2). Let $v = I(s1,s2/C)$. If $v < \epsilon$, go to step 8; otherwise, if $|C|=1$, let $s1$ and $s2$ be parents of the node in C , go to step 8.
6. Let $C'=C$. For each $i \in [1,|C|]$, let $C_i = C \setminus \{the\ i^{th}\ node\ of\ C\}$, $v_i = I(s1,s2/C_i)$. If $v_i \leq v + \epsilon$ then $C' = C' \setminus \{the\ i^{th}\ node\ of\ C\}$, let $s1$ and $s2$ be parents of the i^{th} node of C if the i^{th} node is a neighbor of both $s1$ and $s2$. If $v_i < \epsilon$, go to step 8. (ϵ is a small value.)
7. If $|C'| < |C|$ then let $C=C'$, if $|C'| > 0$, go to step 5.
8. Go back to step 1 and repeat until all pairs of nodes are examined.
9. For any three nodes a, b, c , if a is a parent of b , b and c are adjacent, and a and c are not adjacent and edge (b, c) is not oriented, let b be a parent of c .
10. For any edge (a, b) that is not oriented, if there is a directed path from a to b , let a be a parent of b .
11. Go back to step 9, and repeat until no more edges can be oriented.

In step 1, the procedure tries to find a pair of nodes that may be the endpoints of a V-structure. It then uses steps 2 to 7 to identify colliders between the two nodes. This process continues until all pairs of nodes have been examined. In steps 9 to 11, the procedure uses the identified colliders to infer the directions of other edges. The inference procedure applies two rules: (1) If an undirected edge belongs to a V-structure and the other edge in the structure is

pointing to the mid-node, we orient the undirected edge from the mid-node to the end node. (Otherwise, the mid-node would be a collider and this should have been identified earlier.) (2) For an undirected edge, if there is a directed path between the two nodes, we can orient the edge according to the direction of that path. These two rules are the same as those used in all other collider identification based methods.

The correctness proof of this procedure is given in Section 4.4.

4.4 Correctness Proofs

In addition to the three assumptions made in Section 2.2, we also assume that the given data set has a monotone DAG-faithful probabilistic model before we prove that Algorithm B can always find exactly the edges of the underlying model. The definition of monotone DAG-faithful is given in Definition 4.1.

Definition 4.1 A **monotone DAG-faithful** model is a DAG-faithful probability model in which, for any two nodes that are connected by at least two adjacency paths, by using any cut-set that will not open some paths and close some other paths simultaneously, the mutual information increases if and only if the cut-set opens some previously closed paths between the two nodes.

In real world situations most DAG-faithful models are also monotone DAG-faithful. We conjecture that the violations of monotone DAG-faithfulness only happen when the probability distributions are ‘near’ the violations of DAG-faithfulness. In such situations, other algorithms also have difficulties to generate the true underlying model.

Given a data set that has a monotone DAG-faithful underlying model and satisfies the three assumptions of Section 2.2, Algorithm B can always generate the exact same edges as those of the underlying dependency model M . We prove the correctness of the algorithm by the following propositions.

Proposition 4.1 *Graph G_2 generated after Phase II contains all the real edges of M .*

Proof: Phase I and Phase II of our algorithm examined all the edges between any two nodes that are not independent. An edge is *not* added only if the two nodes are separated by a set of other nodes. Hence, any two nodes that are not directly connected in G_2 are conditionally independent in M . Q.E.D.

Lemma 4.1 *In Procedure `try_to_separate_B` of Phase III, by using the initial condition-set N_1+N_1' or N_2+N_2' , we can close all the paths of the underlying model M between $node_1$ and $node_2$ except the paths connecting $node_1$ and $node_2$ by one collider.*

Proof: By using N_1+N_1' or N_2+N_2' as the condition-set, we instantiate the nodes in N_1 or N_2 (the neighbors of $node_1$ or $node_2$ on the paths between $node_1$ and $node_2$) and N_1' or N_2' (the neighbors of nodes of N_1 or N_2 that are on the paths between $node_1$ and $node_2$). Therefore, at least two consecutive nodes of any path that has length equal to or larger than three are instantiated. Because two consecutive nodes of a path cannot both be colliders in the path, and all the paths of the underlying model M are in the current graph, we can close all the paths in M between $node_1$ and $node_2$ that have length equal to or larger than three. The only open paths are those connecting $node_1$ and $node_2$ by one collider. Q.E.D.

Lemma 4.2 *Procedure `try_to_separate_B` does not open any previously closed path by removing a node from condition-set C .*

Proof: To prove this lemma, it is sufficient to prove that removing a node from C cannot open some paths and close other paths at the same time. If removing a node can only open some paths, by the assumption of *monotone DAG-faithful*, it must increase the mutual information and the procedure will not remove such node. Therefore, the paths cannot be opened. Now, we will prove that removing a node from C cannot open and close paths simultaneously. From lemma 4.1 we know that initially the only open paths are those connecting $node_1$ and $node_2$ by one *collider*. For a node in C that is not a child-node of both $node_1$ and $node_2$ or a descendent of such a child-node, removing it may open some paths but cannot close the paths connecting $node_1$ and $node_2$ by a collider. For a node v in C that is a child-node of both $node_1$ and $node_2$ or a descendent of such child-node, if one of the descendents of v are in C , then removing v cannot close the path connecting $node_1$ and $node_2$ by the child-node. If none of its descendents are in C , removing v may close the path connecting $node_1$ and $node_2$ by the child-node but cannot open a path because the would-be opened path must go through a collider that is a descendent of v . Since none of the descendents of v is in C , such a path cannot be opened. Q.E.D.

Lemma 4.3 *Procedure `try_to_separate_B` can remove all the descendents of both $node_1$ and $node_2$ from condition-set C .*

Proof: Suppose S is a subset of set C and all the nodes in S are the descendants of both $node1$ and $node2$ and cannot be removed, then there must exist a node $v \in S$ which is not an ancestor of any other nodes in S . Since v cannot be removed, removing it must increase mutual information. From the assumption of monotone DAG-faithfulness and Lemma 4.2, we know that removing v will open at least one path. Therefore, node v is not a collider in such path and such path must go through at least one descendent of v . Because a descendent of v is also a descendent of both $node1$ and $node2$, there must exist at least one descendent of v which is a collider in such open path. To make such path open, this collider has to be in S . This contradicts our assumption that v is not an ancestor of any other nodes in S .

Q.E.D.

Proposition 4.2 *Given that graph G contains all the edges of a probabilistic model M , if two nodes a and b are independent in M , Procedure `try_to_separate_B` can always separate them in G .*

Proof: From lemma 4.1 we know that initially the only open paths are those connecting node a and b by one collider. From lemma 4.2 and lemma 4.3 we know that the procedure does not open any path when removing nodes from C and all the descendants of both a and b in C are removed. Therefore, if node a and node b are independent in M , Procedure `try_to_separate_B` can separate them by closing all the open paths. Q.E.D.

Proposition 4.3 *Graph $G3$ generated after Phase III contains the exact same edges as those of M .*

Proof: Since $G2$ contains all the edges of M , and an edge is removed in Phase III only if the pair of nodes are conditionally independent in M , $G3$ also contains all the edges of M . From Proposition 4.2, we also know that if two nodes are independent in M , our algorithm can always separate them in $G3$. Hence, $G3$ contains the exact same edges as those of M . Q.E.D.

Proposition 4.4 *Given that graph G contains the exact same edges as those of the underlying model M , all the colliders that can be identified by Procedure `orient_edges` (G) are the real colliders of M .*

Proof: For any structure $s1-a-s2$ and $s1$ and $s2$ are not directly connected, Procedure `orient_edges` (G) uses steps 1 to 7 to check if a is a collider on the path $s1-a-s2$. Because steps 1 to 7 of this procedure are virtually the same as Procedure `try_to_separate_B`, from Proposition 4.2 we know that it can identify a collider correctly. Since there are no pseudo-edges in G , step 6 of this procedure can never orient an edge wrongly. It is also easy to see that the inference of steps 9 to 11 of the procedure is correct. Q.E.D.

4.5 Complexity Analysis

As we discussed in Section 3.4, the number of CI tests used in a certain dependency analysis based algorithm is a good index for evaluating the algorithm's efficiency. In this section, we analyze the time complexity of Algorithm B on CI tests. The time complexity of each CI test on basic operations is presented in Section 3.4.

Time complexity of Phase I: Phase I computes mutual information between any two nodes; it needs $N(N-1)/2$ (N is the number of nodes) mutual information computations. (A mutual information computation can be viewed as a CI test with an empty condition-set.) Therefore, this phase requires $N(N-1)/2$ CI tests.

Time complexity of Procedure `try_to_separate_A`: In the worst case, $N1$ and $N2$ both contain $N-2$ nodes. This procedure first performs one CI test using $N1$ or $N2$ as the initial condition-set. Then it performs $N-2$ CI tests by using the $N-2$ subsets of cardinality $N-3$ of the initial condition-set as condition-sets. Suppose the procedure can never separate the two nodes, this procedure will remove one node permanently by choosing one subset among those $N-2$ subsets and repeat the last step until the cardinality of the condition-set is equal to one. Therefore, the procedure will perform $1 + (N-2) + (N-3) + \dots + 2 = (N-1)(N-2)/2$ CI tests starting from one of the two set $N1$ and $N2$. Using both $N1$ and $N2$, the total number of CI tests used in the worst case should be less than $(N-1)(N-2)$ since many CI tests are the same when starting from $N1$ and $N2$. Hence, the complexity of this procedure is $O(N^2)$ on CI tests.

Time complexity of Phase II: This phase tries to add edges to the graph we get from Phase I. Since there are at most $N(N-1)/2 - (N-1)$ edges to be added, this phase will call Procedure `try_to_separate_A` at most $N(N-1)/2 - (N-1)$ times. An execution of Procedure `try_to_separate_A` requires at most $(N-1)(N-2)$ CI tests. Therefore, Phase II requires at most $O(N^4)$ CI tests.

Time complexity of Procedure *try_to_separate_B*: This procedure is quite similar to Procedure *try_to_separate_A*. In the worst case, the initial condition-set contains $N-2$ nodes. Suppose each iteration can only remove one node from further trials, the total number of CI tests required is $(N-1)(N-2)/2$.

Time complexity of Procedure *orient_edges*: This procedure examines each pair of nodes to see if there are colliders between them – i.e., if the two nodes and a mid-node form a V-structure of the third type. There are at most N^2 such examinations. The time complexity of each such examination is the same as the time complexity of an execution of Procedure *try_to_separate_B*. Therefore, the time complexity of this procedure is $O(N^4)$ on CI tests.

Time complexity of Phase III: This phase tries to remove each edge from the graph we get from Phase II. Since there are at most $N(N-1)/2$ edges in the graph, this phase will use Procedure *try_to_separate_A* and Procedure *try_to_separate_B* at most $N(N-1)/2$ times. Since an execution of Procedure *try_to_separate_A* and Procedure *try_to_separate_B* requires CI tests at most $O(N^2)$ times, the total number of CI tests of this phase before edge orientation is of complexity $O(N^4)$. Because Procedure *orient_edges* requires CI tests $O(N^4)$ times. The complexity of Phase III is $O(N^4)$ on CI tests.

Overall, Algorithm B requires CI tests at most $O(N^4)$ times.

4.6 Discussion

Quantitative Conditional Independence Test

Algorithm B extends Algorithm A by dropping the requirement of node ordering. Given a data set that has a monotone DAG-faithful underlying model, the algorithm can generate a graph that has the same edges as those of the underlying model using only $O(N^4)$ number of CI tests. It can also orient many of the edges of the graph correctly. Under the same condition, all other dependency analysis based algorithms, like the SGS algorithm [Spirtes *et. al.*, 1990], the Verma-Pearl algorithm [Verma and Pearl, 1992] and the PC algorithm [Spirtes and Glymour, 1991], require exponential numbers of CI tests.

The ability of avoiding exponential complexity on CI tests is due to the two techniques we developed, the three-phase mechanism and the quantitative CI test method. Exponential complexity is unavoidable without (any one of) them. The three-phase mechanism allows us to learn the correct structure by making $O(N^2)$ node-separation decisions without using high order CI tests. However, if we only use CI tests qualitatively, each such decision will require an exponential number of CI tests. In Section 4. 2, we showed how our quantitative CI test method reduced the complexity of each such decision to $O(N^2)$. As a result, the total number of CI tests required in Algorithm B is of $O(N^4)$.

In this algorithm, we use conditional mutual information tests as quantitative CI tests. However, it is also possible to use the Pearson chi-squared tests and the likelihood-ratio chi-squared tests as quantitative CI tests. In fact, mutual information tests and the likelihood-ratio chi-squared tests are highly related. They only differ by a constant $2N$, where N is the number of cases in the data set. Some research [Spirtes, *et. al.*, 1996] shows that the likelihood-ratio chi-squared test and the mutual information test often lead to better results than the Pearson chi-squared test in practice. The reason that we view Bayesian network learning from an information theoretic perspective is that we think it is natural and convenient to present our algorithms this way. Moreover, by using information theoretic measures, we can easily relate our algorithms to entropy scoring and MDL based algorithms like the BENEDICT Algorithm [Acid and Campos, 1996b] and the Lam-Bacchus Algorithm [Lam and Bacchus, 1994], which will be introduced in Section 6. One of our further research directions is to combine our approach with the cross entropy or MDL based scoring approach.

Incorporating Domain Knowledge

In Section 3.5, we already introduced the seven types of domain knowledge for Bayesian network learning and we discussed the method of incorporating domain knowledge into Algorithm A. This method can be used readily in Algorithm B as well. Besides, since the complete node ordering is not given, partial node ordering can then be used

in Algorithm B to improve its efficiency. Partial node ordering is a set of pairs of nodes. Each such pair declares that which of the two nodes should appear earlier than the other in a correct ordering. Obviously, these relations can help us to orient some edges in the first and second phases so that the edge orientation procedure at the end of the third phase can be finished more quickly. These relations can also be used in several other parts of the algorithm to improve performance. For example, in Procedure *try_to_separate_A*, we need to find $N1$ and $N2$, which are the neighbor-sets of *node1* and *node2* respectively. Since the procedure tries to separate the two nodes using only the parents of *node1* (*node2*) in $N1$ ($N2$), if we know that some nodes in $N1$ ($N2$) that are actually the children of *node1* (*node2*), we can remove them immediately without using any CI tests. This will improve both the efficiency and accuracy of this procedure.

In general, as we stated in Section 3.5, our algorithms can be viewed as constraint based algorithms. Therefore, domain knowledge can be readily used as constraints on learning Bayesian networks.

Monotone DAG-Faithfulness

Unlike Algorithm A, which requires the assumption of DAG-faithfulness for its correctness proof, Algorithm B requires a stronger assumption, called the monotone DAG-faithfulness assumption. It assumes that the underlying probabilistic model of a data set is monotone DAG-faithful. The definition of monotone DAG-faithfulness was given in Section 4.4.

From the definition of monotone DAG-faithful models we know that these models form a subset of DAG-faithful models. We have found that some models are DAG-faithful but not monotone DAG-faithful. To illustrate the relationship between DAG-faithfulness and monotone DAG-faithfulness, we use the following probabilistic model.

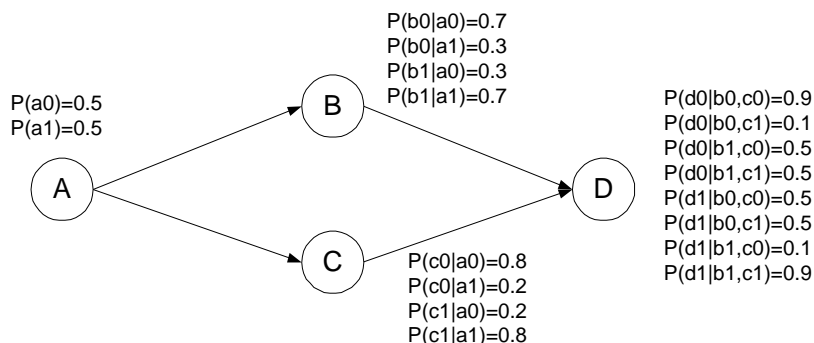


Figure 4.4

In the above graph, we present a complete Bayesian network, which includes a DAG structure and four CP tables for the four nodes. Using Equation (2.2), we can calculate the conditional mutual information of the pairs of nodes from the CP tables. Let us focus on the conditional mutual information between node B and node C . Using a computer program, we get that $I(B, C | A, D) = 0.018$ and $I(B, C | D) = 0$. This clearly contradicts our intuition since when using $\{A, D\}$ as the condition-set, there is one open path $B-D-C$ and when using $\{D\}$ as the condition-set, there are two open paths $B-D-C$ and $B-A-C$. If the model was monotone-DAG-faithful, $I(B, C | D)$ should have been greater than $I(B, C | A, D)$. Please note that this model is not even DAG-faithful since the independence between B and C given $\{D\}$ cannot be expressed by the DAG structure. However, if we change the parameters of the network a little, for instance, changing the CP table of node C to the same as that of node B , we can make $I(B, C | D)$ greater than 0 but still smaller than $I(B, C | A, D)$. Now, we get a model that is DAG-faithful since B and C are not independent given $\{D\}$, but not monotone DAG-faithful.

From the above example, we can draw two conclusions. Firstly, there exist some models that are DAG-faithful but not monotone DAG-faithful. Secondly, the distinction between DAG-faithful models and non-DAG-faithful models is not black and white. In the above example, if the small value $I(B, C | D)$ happens to be larger than the threshold used to separate ‘dependent’ and ‘independent’, then the model is DAG-faithful; otherwise, it is not DAG-faithful. This shows that there is a ‘gray’ area of the area of DAG-faithful models, in which the models are ‘close’ to being non-DAG-faithful. Although we do not have a formal proof, we conjecture that the non-monotone DAG-faithful models are all in the ‘gray’ area. In other words, if a model violates the monotone DAG-faithfulness

assumption, we conjecture that it is also close to the violation of DAG-faithfulness. So, this model may also cause problems to other algorithms.

Given the fact that qualitative CI test based methods, like Algorithm A, require the qualitative DAG-faithfulness assumption for their correctness proof, it is reasonable to think that our quantitative CI test based method requires a quantitative assumption. We view the monotone DAG-faithfulness assumption used in Algorithm B as the quantitative counterpart of the DAG-faithfulness assumption. We believe that most real-world probabilistic models are actually monotone DAG-faithful.

Even when the underlying probability distribution is DAG-faithful but not monotone DAG-faithful, this algorithm may still be able to learn the correct graph. In fact, this algorithm may not be able to separate two d-separated nodes only when there is at least one path that connects the two nodes by a single collider and removing a node in the condition-set causes the violation of the monotone DAG-faithful assumption. However, since this will only cause one edge to be wrongly added to the current graph, the correctness of other edges in the graph will not be affected and the resulting graph can still be very close to the real model.

Although we think it is unnecessary, if it is really needed to learn the correct graph for all DAG-faithful models, we can always replace Procedure *try_to_separate_B* (*current graph*, *node1*, *node2*) with a procedure that tries to separate the two nodes by using every subset of the neighbors of *node1* and then every subset of the neighbors of *node2*. This procedure is presented as follows.

Procedure *try_to_separate_B** (*current graph*, *node1*, *node2*)

1. Find the neighbors of *node1* and *node2* that are on the *adjacency* paths between *node1* and *node2*. Put them into two sets *N1* and *N2* respectively.
2. Remove the currently known child-nodes of *node1* from *N1* and child-nodes of *node2* from *N2*.
3. If the cardinality of *N1* is greater than that of *N2*, swap *N1* and *N2*.
4. Let *C* equal to *N1*.
5. For every subset C_i of *C*, let $v_i = I(\text{node1}, \text{node2} / C_i)$. If $v_i < \epsilon$, return ('separated').
6. If *N2* has not been used, let *C* equal to *N2* and go to step 5; otherwise, return ('failed').

By the definition of DAG-faithfulness, this procedure is always correct. Theoretically, it requires an exponential number of CI tests in the worst case. However, for a sparse network, it will not influence the performance very much since this procedure is only used at the end of the third phase. The experimental results show (Section 6) that most running time is consumed in Phase I and Phase II.

5 Empirical Study

In this section, we first use three Bayesian networks to empirically evaluate Algorithm A and Algorithm B. Then, we give a brief introduction to our Bayesian network learning tool – PowerConstructor, which implemented both Algorithm A and Algorithm B.

The three Bayesian networks are the ALARM network, which has 37 nodes and 46 arcs, the Hailfinder network, which has 56 nodes and 66 arcs, and the Chest-clinic network, which has 8 nodes and 8 arcs. The first two networks are from moderate complex real-world domains and the third one is from a simple fictitious medical domain. All three data sets are synthesized from their underlying probabilistic models using a *Monte Carlo* technique, called *probabilistic logic sampling* method [Henrion, 1988]. Although we have applied our system into a real-world application (a telecommunication fault diagnosis system) with success, and from personal communications we know that our system has also been used in several real-world applications by other researchers (Unfortunately, we do not have their data sets.), we found it is difficult to use real-world data to analyze our algorithms. The reason is that in most real-world data sets, we do not know their underlying probabilistic models, and therefore it is difficult to evaluate the experimental results. On the contrary, if we have a 'golden' Bayesian network at hand, we can easily evaluate the accuracy of the experimental results. In fact, almost all researchers in this area use synthesized data sets to evaluate their algorithms.

When performing these experiments, we run our PowerConstructor in *debug* mode so that we can monitor the intermediate results and the number of CI tests used. Therefore, the running times we give in this section are slightly longer than the actual running time of our system when running under the *normal* mode. All these experiments were conducted on a Pentium 133 MHz PC with 32 MB of RAM running under Windows NT 4.0. The data sets are stored in an MS-Access database.

1998]). Because of the lack of space, we only give our results on one version of the ALARM network in this paper. In Sections 5.1.1 and 5.1.2, we give our detailed results on 10,000 cases of the ALARM network using Algorithm A and Algorithm B, respectively. In Section 5.1.3, we give our results on different sample sizes of the ALARM network using both algorithms.

5.1.1 An Experiment on the ALARM Network Data Using Algorithm A

In the following table, we summarize our result of each phase in terms of the structure learned, the number of CI tests used and the running time. The learned structure is evaluated by using the number of missing arcs and wrongly added arcs compared to the true structure; the CI tests are grouped by the cardinalities of their condition-sets.

Phase	Results			No. of CI Tests						Time (Seconds)
	Arcs	M.A.	E.A.	0	1	2	3	4+	Total	
I	43	5	2	666	0	0	0	0	666	607
II	50	1	3(2+1)	0	116	54	22	10	202	217
III	45	1	0	0	12	1	1	3	17	20

Table 5.1 Results on dataset1 of ALARM network using Algorithm A

(M.A. and E.A. stand for missing arcs and extra arcs respectively. The CI tests are grouped by the cardinalities of their condition-sets.)

From Table 5.1 we can see that using 10,000 cases, Algorithm A can learn a nearly perfect structure (with only one missing arc) in less than 15 minutes on an ordinary PC. The result after Phase I already resembles the true structure to some extent, the draft has 43 arcs, among which 2 arcs should not have been added. The draft also missed 5 arcs of the true structure. The result after the second phase (the thickening phase) has 50 arcs, which includes all the arcs of Phase I and 4 out of 5 of the previously missing arcs. In addition, it also wrongly added another arc. This is quite usual since it is always possible to add some arcs wrongly before all the real arcs are discovered. Theoretically, there should be no missing arcs after Phase II. However, the actual dependency of the two nodes of this missing arc is very weak in the 10,000 cases and this pair of nodes is not included in the list L by step 2 of the algorithm. In Phase III, the algorithm ‘thinned’ the structure successfully by removing all three previously wrongly added arcs. The result after the third phase has 45 arcs, all of which belong to the true structure.

From the complexity analysis of Section 3.4 we know that all of the three phases require $O(N^2)$ number of CI tests in the worst case. However, the number of CI tests used in one phase is quite different from that of another in this experiment. This is because that the ALARM network is a sparse network and Phases II and III only require a large number of CI tests when a network is densely connected, whereas Phase I always requires $O(N^2)$ CI tests. All our experiments on real-world data sets show similar patterns – i.e., most CI tests are used in Phase I.

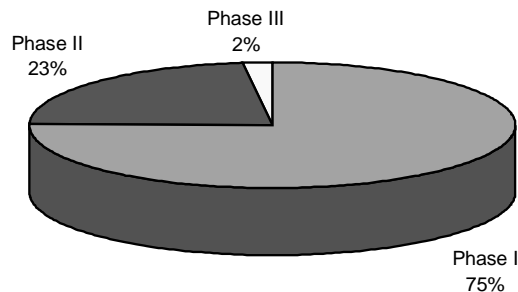


Figure 5.2 The number of CI tests used in each phase.

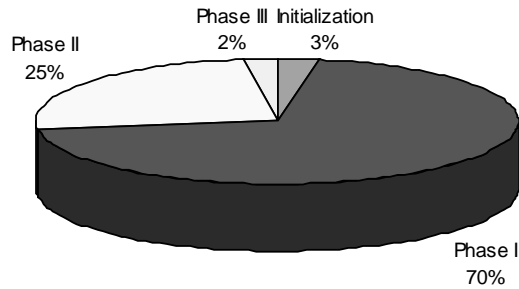


Figure 5.3 The running time used by each phase.

From Figure 5.2 we can see that 75% of the total CI tests are used in Phase I. The reason that we are concerned with the number of CI tests used is that the number of CI tests is highly related to the running time. In this experiment, we found that 861 seconds out of the total 867 seconds of running time (about 99.3% of the running time) is actually used by the CI tests. (Most running time in the initialization procedure is also used for the preparation of CI tests, so we count this running time into the cost of CI tests.) This is why that the percentage of running time of each phase (shown in Figure 5.3) is consistent to the percentage of the total number of CI tests of each phase. Note that Phases II and III take a bit more running time than they should if purely based on the number of CI tests used. The reason is that a high order CI test takes slightly longer to finish than a low order CI test. In this experiment, a CI test with empty condition-set takes on average about 0.9 second to finish; with condition-set of one node, two nodes and three nodes, a CI test takes 1.0 second, 1.1 second and 1.2 second, respectively. This is one of the reasons that we are also concerned with the cardinalities of the condition-set of the CI tests. The other reason is that a CI test can be unreliable if the cardinality of its condition-set is too high. In this experiment, the largest condition-set contains only five variables.

Since CI tests consume most of the running time in Bayesian network learning, to speed up the learning process, we need to speed up each CI test. After further analysis, we found that data base operations consume most of the running time in each CI test. Therefore, we can improve the efficiency of our program by using high performance database query engines. To prove this, we moved this data set to a remote ODBC database server (SQL-server 6.5 running under Windows NT Server 4.0 on a Dual Pentium 166 MHz computer) and repeated the experiment. We found that the experiment can be finished 13% faster. Note that our system was still running on the local PC as before.

5.1.2 An Experiment on the ALARM Network Data Using Algorithm B

Phase	Results			No. of CI Tests						Time (Seconds)
	Edges	M.E.	E.E.	0	1	2	3	4+	Total	
I	36	12	2	666	0	0	0	0	666	607
II	49	2	5(2+3)	0	127	61	22	7	217	239
III	44	2	0	0	86	6	8	3	103	109

Table 5.2 Results on dataset1 of ALARM network (Algorithm B)

(M.E. and E.E. stand for missing edges and extra edges respectively. The CI tests are grouped by the cardinalities of their condition-sets.)

From Table 5.2 we can see that (using 10,000 cases) Algorithm B can get a very good result (with only two missing edges) in about 16 minutes. The result after Phase I (drafting) has 36 edges, among which 2 edges are wrongly added. The draft also missed 12 edges of the true structure. It is not very satisfactory compared to the draft we get using Algorithm A (Section 5.1.1), which only misses 5 edges. However, this is due to the difference in Phase I of the two algorithms. In this algorithm, since the node ordering is not know, the Phase I stops at a draft that has a polytree structure. The result after the second phase (the thickening phase) has 49 edges, which includes all the edges of Phase I and 10 out of 12 of the previously missing edges. This algorithm cannot discover the other two edges due to fact that the two relationships are too weak. In addition, it also wrongly added 3 edges. In Phase III, the algorithm ‘thinned’ the structure successfully by removing all five wrongly added arcs. The algorithm can also orient 40 of the 44 learned edges correctly. It cannot orient the rest 4 edges due to the limitation of collider

identification based method and no other algorithm based on the same method can do better. By comparing Table 5.2 to Table 5.1, we can see that the results of the three phases of Algorithm B are not as good as those of Algorithm A, and Phase II and Phase III of Algorithm B require more CI tests and are slower than the two corresponding phases of Algorithm A. This is not surprising since the Algorithm B does not use the node ordering as prior knowledge as Algorithm A does.

From the complexity analysis of Section 4.5, we know that the first phase requires N^2 CI tests and the second and the third phases are of the complexity $O(N^4)$ in the worst case. However, since most real-world situations have sparse Bayesian networks, the numbers of CI tests used in the second and the third phases are usually much smaller than the number of CI tests used in the first phase, which is of complexity $O(N^2)$. As in Section 5.1.1, we use two pie charts to show the percentages of the number of CI tests used and the running time used of each phase respectively.

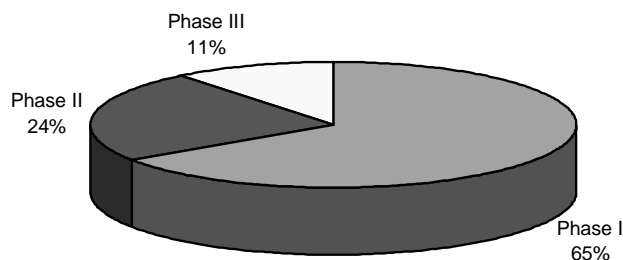


Figure 5.4 The number of CI tests used at each phase.

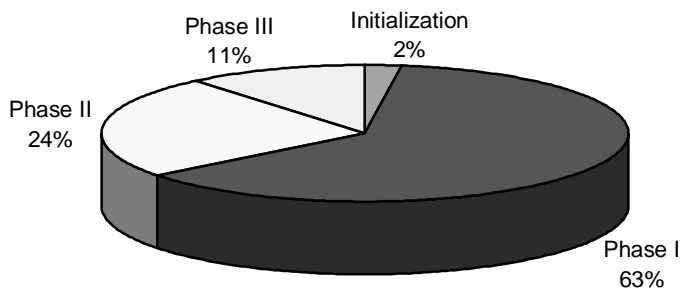


Figure 5.5 The running time used by each phase.

From the above two pie charts we can see that Phase I uses much more CI tests than Phase II and Phase III, and it also consumes much more running time than the other two phases. However, by comparing the two pie charts with the two pie charts in Figures 5.2 and 5.3, we can see that the “shares” given to Phase I of this algorithm in both pie charts are smaller than those of Algorithm A, and the shares of Phase III are larger than those of Algorithm A. This is due to the fact that the Phase I in both algorithms require the same number of CI tests whereas other phases of Algorithm B require more CI tests than in Algorithm A. Because Phase III of this algorithm also need to use a procedure to orient the edges, it uses significantly more CI tests and is much slower than Phase III of Algorithm A.

5.1.3 Experiments on Different Sample Sizes

In this section, we present our experimental results on 1,000, 3,000, 6,000 and 10,000 cases of the ALARM network data. The results are shown in the following table.

Cases	Ordering	Results				Time (Seconds)
		M.E	E.E.	M.O.	W.O.	
1,000	Yes	0	3	N/A	N/A	112
	No	3	2	3	2	127
3,000	Yes	1	3	N/A	N/A	273
	No	1	1	4	0	320
6,000	Yes	1	0	N/A	N/A	543
	No	2	0	4	0	607
10,000	Yes	1	0	N/A	N/A	867
	No	2	0	4	0	978

Table 5.3 Results on 1,000, 3,000, 6,000 and 10,000 cases

To see the trends in the above table, we give the following chart.

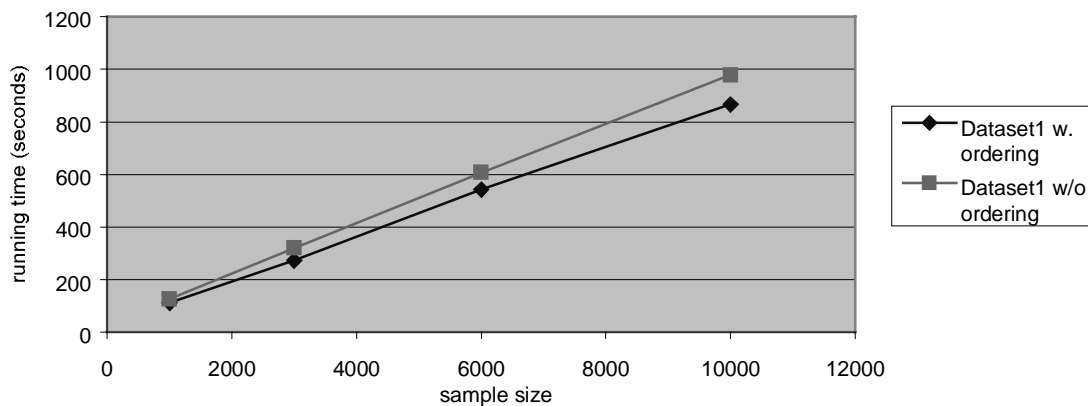


Figure 5.6 The relationship between the sample sizes and the running time

From Figure 5.6 we can see that the running time is roughly linear to the number of cases in the data set. This is what we expected since most of the running time of the experiments is consumed by database queries; and response time of each database query is roughly linear to the number of records in the database table. This shows that our algorithms are capable of handling much larger data sets since the running time won't increase too fast. From Table 5.4 we can also see that the general trend is that the larger the data sets are the fewer the errors. However, the results on 3,000 cases of these data sets are already quite acceptable. This shows that our algorithms can give reliable results even when the data set is not large for its domain. The reason is that our algorithms can avoid many high order CI tests, which are unreliable when the data sets are not large enough.

5.2 The Hailfinder Network

Hailfinder network is another real-world domain belief network. It is a normative system that forecasts severe summer hail in northeastern Colorado. The network structure is shown in Figure 5.7, which contains 56 nodes and 66 arcs. Each node (variable) has from two to eleven possible values. (Notice that each node in the ALARM network only has from two to four possible values.) For detailed information about the Hailfinder project and various related documents, please visit the web page <http://www.sis.pitt.edu/~dsl/hailfinder>.

To evaluate our algorithms, we generated a data set of 20,000 cases from the underlying probabilistic model of Hailfinder network (version 2.5) using the probabilistic logic sampling method introduced earlier in this section. This sampling process took about one hour on a Pentium 133 MHz PC to complete.

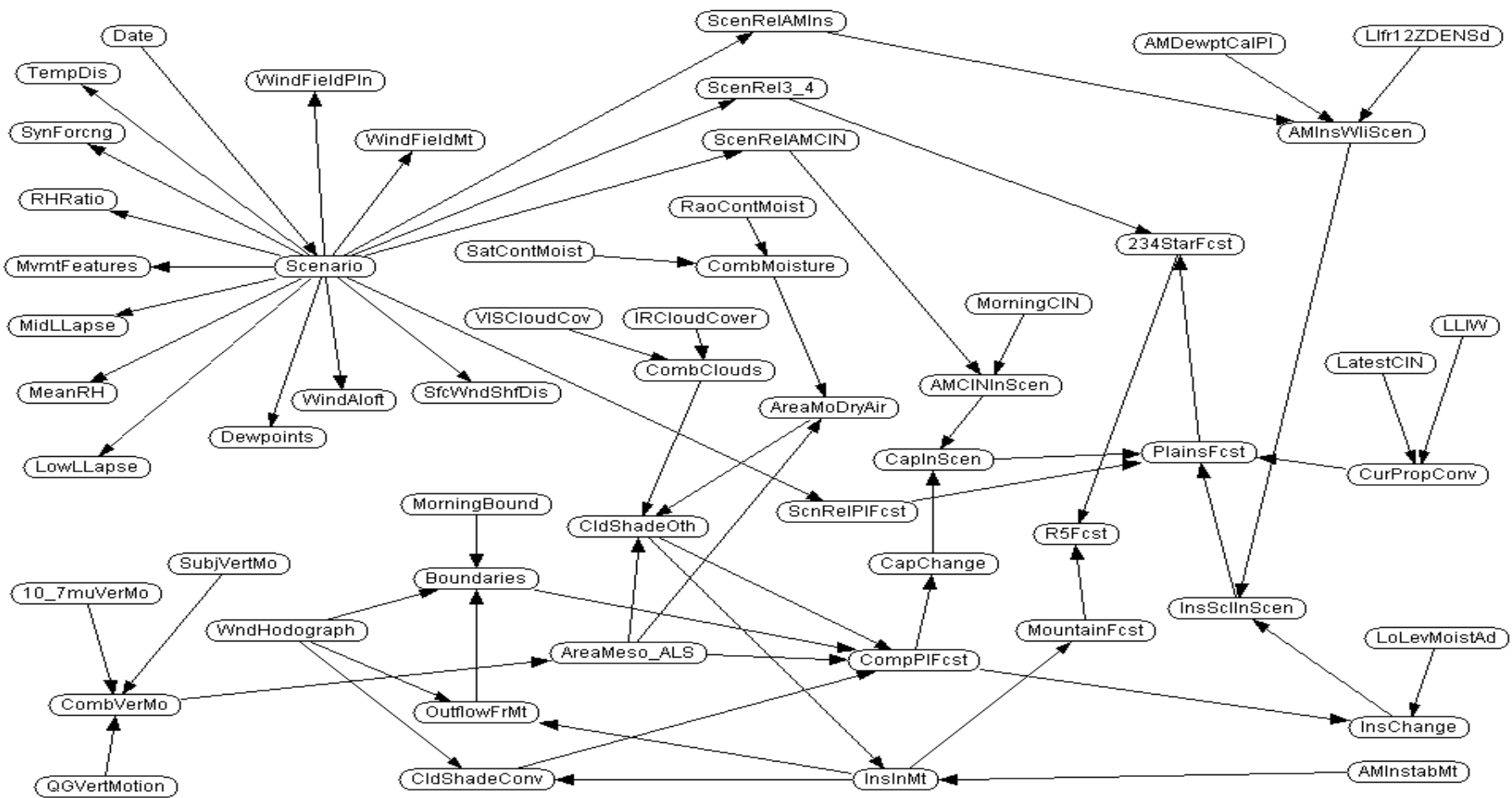


Figure 5.7 The Hailfinder network

5.2.1 Experiments on 10,000 cases

In this section, we give the detailed experimental results using the first 10,000 cases of the Hailfinder data. The results are from two runs of our system, one with node ordering (Algorithm A), and the other without node order (Algorithm B). The node ordering we used for Hailfinder network is the ordering described in the file <http://www.sis.pitt.edu/~dsl/hailfinder/hailfinder25.dne>.

Node ordering	Number of CI tests					Running time (seconds)				
	0	1	2	3+	Total	Init	P.I	P.II	P.III	Total
Yes	1540	163	33	7	1743	32	1273	186	17	1508
No	1540	290	18	1	1849	32	1273	211	98	1614

Table 5.4 Running time and the CI tests used on 10,000 cases of Hailfinder data (The CI tests are grouped by the cardinalities of their condition-sets.)

From the above table we can see that when node ordering is given, the running time is about 25 minutes; and when node ordering is not given, the running time is about 27 minutes.

Due to the lack of space, the experimental results are not included in Table 5.4. When node ordering is given, the learned network has three missing arcs and no extra arcs. When node ordering is not given, the learned network has 4 missing edges and 1 extra edges. Among these edges, one is not oriented and five are wrongly oriented.

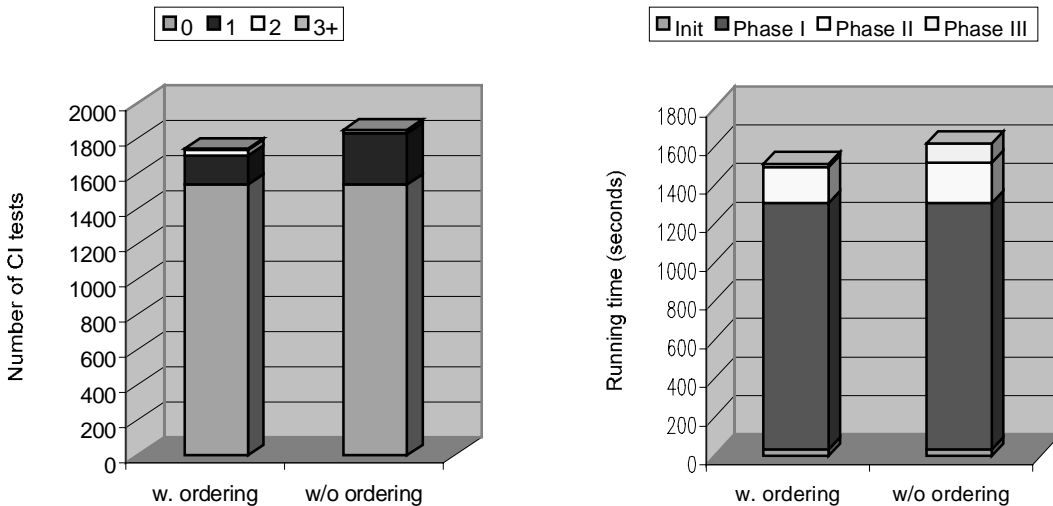


Figure 5.8 The bar charts of CI tests and running time. (The number 0, 1, 2, 3+ in the left bar chart represent the cardinalities of the condition-sets of CI tests.)

The above two bar charts show that most running time is consumed by Phase I and most CI tests are of the low orders. Comparing the number of CI tests used in both algorithms on the Hailfinder data set and the ALARM network data set, we can see that although in theory the complexity of Algorithm B (the case when node ordering is not given) is $O(N^4)$ on CI tests, the actually speed of complexity growth is similar to that of Algorithm A, which is $O(N^2)$ in nature. This shows that in real-world situations, the actual time complexity on CI tests is close to $O(N^2)$ even when node ordering is not given. In these two experiments, we also found that the time cost of each CI test is roughly the same as that in the experiments using 10,000 cases of ALARM data. This shows that the time cost of a CI test is largely determined by the sample size and is not much affected by the number of attributes in the data set.

5.2.2 Experiments on Different Sample Sizes

In this section, we present our experimental results on 2,500, 5,000, 10,000, 20,000 cases of the Hailfinder data. The results are shown in the table below.

Cases	Ordering	Results				Time (Seconds)
		M.E	E.E.	M.O.	W.O.	
2,500	Yes	2	6	N/A	N/A	524
	No	5	4	1	5	549
5,000	Yes	3	3	N/A	N/A	958
	No	3	2	0	2	988
10,000	Yes	3	0	N/A	N/A	1508
	No	4	1	1	5	1614
20,000	Yes	3	0	N/A	N/A	2792
	No	4	1	1	5	3187

Table 5.5 Results on different sample sizes of Hailfinder data

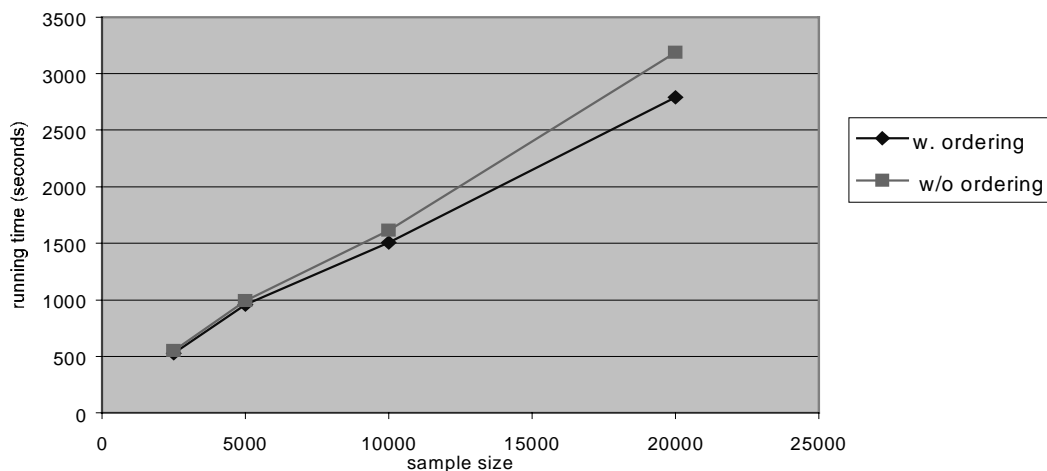


Figure 5.9 The relationship between the sample sizes and the running time

These results on Hailfinder data repeat the trends we found on ALARM data – that is, the growth of running time is roughly linear to the number of cases in the data set, and in general, with the increase of the sample size, the number of errors decreases until the sample size is large enough. From Table 6.7 we can see that the results we get using 10,000 cases is the same as the results we get using 20,000 cases, which shows that 10,000 cases of Hailfinder data is already large enough for our algorithms.

5.3 The Chest-clinic Network

The Chest-clinic network (also known as Asia network) is a very small belief network for a fictitious medical domain about whether a patient has tuberculosis, lung cancer or bronchitis, related to their X-ray, dyspnea, visit-to-Asia and smoking status. The structure of this network is shown in Figure 5.10, which contains 8 nodes and 8 arcs. Each node has only two possible values. The underlying probabilistic distribution of this network is described in <http://www.norsys.com/netlib/Asia.dnet>. We generated a data set of 1,000 cases using the probabilistic logic sampling method. (A fragment of the data set is in Appendix C.) This simple Bayesian network is used to show the performance of our algorithms on small domains.

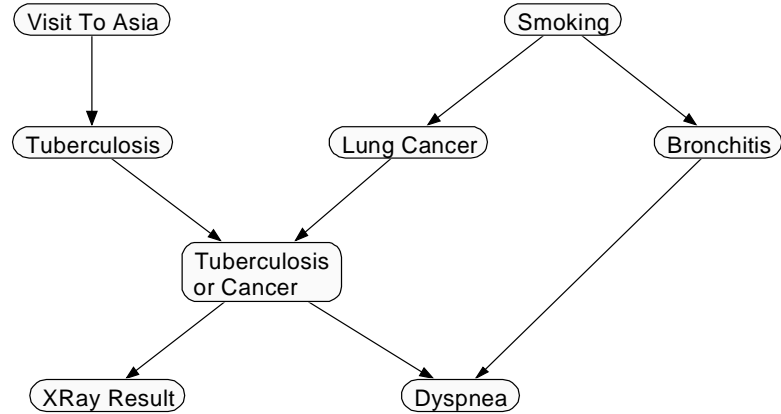


Figure 5.10 The Chest-clinic network

Our results are summarized in the table below.

Node Ordering	Results				Time (seconds)
	M.E.	E.E.	M.O.	W.O.	
Yes	1	0	N/A	N/A	6.0
No	1	0	2	0	6.5

Table 5.6 Results on 1,000 cases of Chest-clinic data

The node ordering we use is [Visit to Asia, Tuberculosis, Smoking, Lung Cancer, Tuberculosis or Cancer, X-ray results, Bronchitis, Dyspnea]. In both experiments, the system could not find the arc from ‘Visit to Asia’ to ‘Tuberculosis’. The reason is that the dependency between the two nodes is too weak in the data set. There are also two edges (Smoking – Lung Cancer and Smoking – Bronchitis) that cannot be oriented using Algorithm B. This is due to the limitation of the edge orientation method we use. For this network structure, no other collider identification based methods can do better.

5.4 Introduction to PowerConstructor

Before our system, quite a few commercial systems and research prototypes have been developed, including TETRAD II [Scheines *et. al.*, 1994], Bayesian Knowledge Discoverer [Ramoni and Sebastiani, 1997], CoCo [Badsberg, 1992], BUGS [Thomas *et. al.*, 1992], BIFROST [Hojsgaard *et. al.*, 1994] and MIM [Edwards, 1995]. However, as far as we know, only TETRAD II can handle a data set at the size of the ALARM network data, which contains 37 variables and 10,000 records. Considering that real-world data sets often contain hundreds of variables and millions of records, the size of the ALARM network data is actually quite moderate. The lack of practicable, easy-to-use systems hinders the real use of Bayesian network learning in the industry. As a result, most industry users are unaware of the current progress in this area. This is partially the reason that the Bayesian network method is not as popular as other methods like neural networks, decision trees in current data mining systems in the industry.

To promote the real use of Bayesian networks and facilitate researchers in related fields, we implemented our algorithms into a Bayesian network learning system. We named the system “Bayesian network PowerConstructor”, which has two components, a user-friendly interface and a construction engine. The current version of our system is 2.0 and it is running under 32-bit windows systems (i.e., Windows 95, Windows 98 and Windows NT) on PCs. The system takes as input a database table and constructs a Bayesian network (both structure & parameters) as output. It also supports domain knowledge as additional input.

Our system is available for download from our web site (<http://www.cs.ualberta.ca/~jcheng/bnpc.htm>). Since October 1997, over 2,000 people have visited

our web sites and over 1000 people have downloaded our system. We are also very glad to know that some users have used it on real-world problems.

Next, we will briefly introduce our Bayesian network learning system. A detailed presentation of the system is given in [Cheng, 1998].

The structure of our system is shown in Figure 5.11, from which we can see that the two components of the system (user interface and construction engine) are in a client-server structure and both of them are connected to different kinds of databases through a standard interface, Data Access Objects.

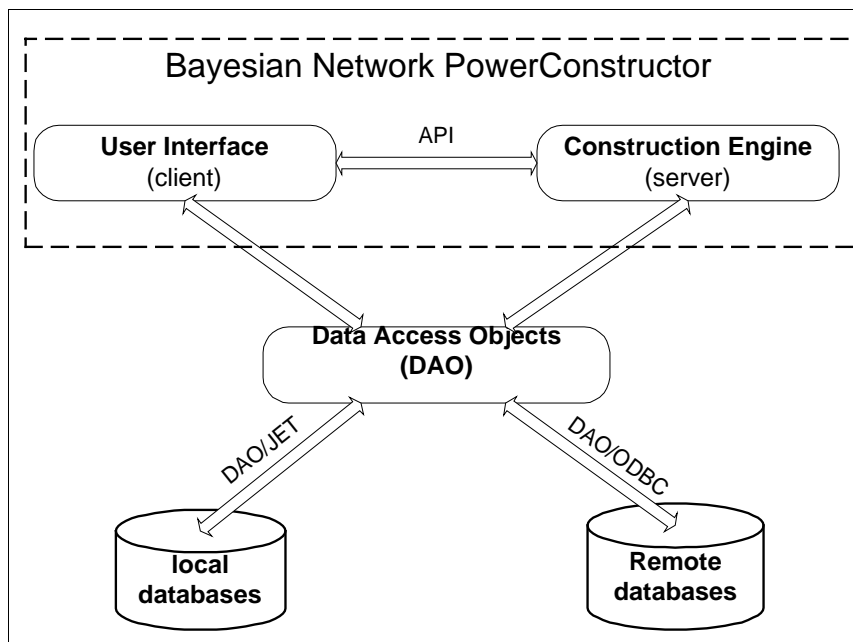


Figure 5.11 System structure of PowerConstructor.

The interface part of PowerConstructor is an executable file (BNPC.EXE). It first gathers the input information from the user using a five-step wizard. This information includes database formats, database location, data set name, domain knowledge etc. Next, the user interface calls the construction engine. When the construction engine finishes the process, the result is passed back as a parameter to the user interface.

The construction engine is an ActiveX code component in the form of a dynamic-link library (DLL) file (BNPCAPI.DLL). By using ActiveX technology into our construction engine and separating the construction engine from the user interface, we made our system easy to maintain and reuse. For instance, we can develop a new user interface or integrate the construction engine into other systems without changing the code of the construction engine at all. Likewise, we can update the construction engine without touching the code of the user interface. The working mechanism of the construction engine is shown in Figure 5.12. For information on how to integrate the construction engine into other system, please refer to the help file comes with our system or [Cheng, 1998].

From Figure 5.11 we can see that both the user interface and the construction engine are connected to the data access objects, which provides a standard interface to access databases. To access local desktop databases like MS-Access, Foxpro and Paradox, we use DAO/JET interface, which provides the database operation functions using the Microsoft Jet database engine. To access remote databases, we use DAO/ODBCDirect interface, which passes commands directly to the remote database servers for processing. Because most of the workload is moved to the high-performance database server, this method can speed up the Bayesian network learning process and save a lot of resources in the local computer. Therefore, it is recommended to store very large data sets in database servers.

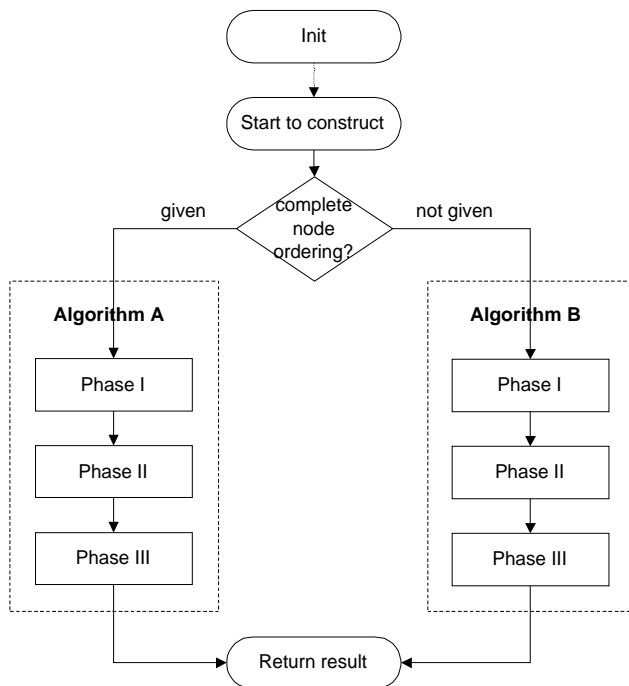


Figure 5.12 Working mechanism of the construction engine.

Compared with other Bayesian network learning systems, our PowerConstructor has the following features.

User Interface:

- Wizard-like interface. It gathers necessary input information through 5 simple steps.
- Online help. Online help is available for each step.
- Graphical belief network editor for modifying BN structure after the learning process.

Construction Engine:

- Accessibility. The system supports most of the popular desktop database and spreadsheet formats, including Ms-Access, dBase, Foxpro, Paradox, Excel and text file formats. It also supports remote database servers like Oracle, SQL-server through ODBC.
- Reusability. The engine is an ActiveX DLL, so it can be easily integrated into other belief network, data mining or knowledge base systems for windows 95/98/NT.
- Efficiency. In theory, it requires CI tests to the complexity of $O(N^4)$ without node ordering and $O(N^2)$ when node ordering is given. (N is the number of attributes.) In practice, the complexity is about $O(N^2)$ even without node ordering.
- Supporting domain knowledge. Complete ordering, partial ordering, direct causes and effects, forbidden links and root & leaf nodes can be used to constrain the search space and therefore speed up the construction process.
- Supporting large data sets. Running time is linear to the number of cases (see Chapter 6).
- Supporting condensed data sets, which has a 'frequency' fields that contains the number of appearances of the current entry in the data.
- Connectivity: The resulting belief network (both structure and parameters) can be exported to other belief network systems. (The current version supports Hugin net format and Netica format.)

6 Related Work

6.1 Overview of Graphical Probabilistic Model Learning Algorithms

Graphical probabilistic models include Bayesian networks and Markov networks. In recent years, graphical model learning has become a very active research topic and many algorithms have been developed for it. For survey papers and introductory papers on probabilistic network learning, please refer to [Buntine, 1996; Chrisman, 1996; Heckerman, 1995; Krause, 1996].

There are two general approaches to graphical probabilistic model learning from data, the *search & scoring* methods and the *dependency analysis* methods. In the first approach, the algorithms view the learning problem as to search for a structure that can fit the data best. They start with a graph without any edges, and then use some search method to add an edge to the graph. After that, they use some scoring method to see if the new structure is better than the old one. If it is, they keep the newly added edge and try to add another one. This process continues until no new structure is better than the previous one. Different scoring criteria have been applied in these algorithms to evaluate a structure, such as Bayesian scoring method [Cooper and Herskovits, 1992; Heckerman *et al.*, 1994; Ramoni and Sebastiani, 1996], entropy based method [Herskovits, 1991], minimum description length method [Suzuki, 1996; Lam and Bacchus, 1994], and minimum message length method [Wallace *et al.*, 1996]. Most of these algorithms apply heuristic search methods. To reduce the search space, many of these algorithms require node ordering. Since learning Bayesian networks using search & scoring methods is NP-hard [Chickering *et al.*, 1994], the application of heuristic search is justifiable.

In the second approach, the learning problem is viewed differently. Since a structure encodes many dependencies of the underlying model, the algorithms of this approach try to discover the dependencies from the data, and then use these dependencies to infer the structure. The dependency relationships are measured by using some kind of CI test. The algorithms described in [Spirtes *et al.*, 1991; Wermuth and Lauritzen, 1983; Srinivas *et al.*, 1990] and the algorithms proposed in this paper belong to this category.

Both of these approaches have their advantages and disadvantages. Generally speaking, the dependency analysis approach is more efficient than the search & scoring approach for sparse networks (the networks that are not densely connected). It can also get the correct structure when the probability distribution of the data satisfies certain assumptions. However, many of these algorithms require an exponential number of CI tests and a lot of high order CI tests (CI tests with large condition-sets). As pointed out in [Cooper and Herskovits, 1992], CI tests with large condition-sets may be unreliable unless the volume of the data is enormous. On the other hand, although the search & scoring approach may not find the best structure due to its heuristic nature, it works with a wider range of probabilistic models than the dependency analysis approach.

In Section 6.2 and Section 6.3, we will introduce some representative algorithms of each group. Some other algorithms are briefly introduced in Section 6.4.

6.2 Search & scoring Based Methods

Chow-Liu Tree Construction Algorithm [Chow and Liu, 1968]

Chow and Liu developed a tree construction algorithm, which has had a far-reaching influence on the area of graphical model learning. It takes a probability distribution P as input and constructs a tree structure as output in only $O(N^2)$ steps. (N is the number of nodes.) They proved that the resulting tree-dependent distribution P^t is the best approximation of P , and when the underlying structure of distribution P is actually a tree, this algorithm guarantees that that tree structure will be found.

An interesting fact about this algorithm is that it has the characteristics of both graphical model learning approaches presented earlier. Although the general idea behind this algorithm is to find a structure with the best score (Kullback-Leibler cross-entropy), it ends up with analyzing the pair-wise dependencies, which is the method used in the dependency analysis approach. The reason is that this kind of pair-wise analysis guarantees the best score and therefore heuristic search is unnecessary.

The merit of this algorithm is that it only needs $O(N^2)$ pair-wise dependency calculations and each calculation uses only second-order statistics. Unfortunately, an equally efficient algorithm is not

possible for constructing multiply connected graphs through dependency analysis, since condition-sets are involved in pair-wise dependency calculations, which means higher order statistics must be used.

Rebane-Pearl Polytree Construction Algorithm [Rebane and Pearl, 1987]

This algorithm is a direct extension of the Chow-Liu algorithm. A *polytree* (also called *singly connected* graph) is a structure that has no loops, so that there is at most one path between any two nodes in the graph. Rebane and Pearl proved that when the probability distribution has a perfect map and the map has a polytree structure, their algorithm can always find this perfect map. They also developed a method for finding the directions of the edges in the graph by identifying colliders. The idea behind this method is used by many other algorithms that are capable of edge orientation.

K2 Algorithm [Cooper and Herskovits, 1992]

K2 is a representative of search & scoring based algorithms for general Bayesian belief network learning. It takes a data set and a node order as input and constructs the belief network structure as output. The K2 algorithm applies a Bayesian scoring method. The aim of K2 is to find the most probable belief network structure B_s given a data set D , that is, maximizing the probability $P(B_s | D)$. This algorithm is quite famous due to its accurate results on the ALARM network data set, which is a widely accepted benchmark for belief network learning algorithms. In about 17 minutes, this algorithm can generate the network structure with one missing arc and one extra arc from a data set of 10,000 cases on a Macintosh II computer.

HGC (Heckerman, Geiger and Chickering) Algorithm [Heckerman *et. al.*, 1994]

This algorithm is another Bayesian score based algorithms. The significance of this work is that by studying the consistent properties and assumptions of scoring methods, they found two assumptions, called parameter modularity and event equivalence, which have been ignored by other researchers. By using these assumptions with assumptions made by other researchers, Heckerman *et. al.* show that a straightforward method for combining users knowledge and statistical data can be obtained.

Kutato Algorithm [Herskovits and Cooper, 1991]

Instead of using Bayesian scores as in K2 and HGC algorithm, Herskovits and Cooper developed an algorithm that uses entropy measure. They view the learning problem as approximating the true joint probability function of data using a belief network structure that has minimum information loss (maximum entropy). The search method used in this algorithm is similar to the one used in K2. Like K2, this algorithm also requires the node ordering. The time complexity of this algorithm on the number of score calculations is $O(N^4)$ where N is the number of nodes in the network. This algorithm has also been tested using ALARM network data, in approximately 22.5 hours on a Macintosh II computer, it can generate a network structure from 10,000 cases of the data set with 2 missing arcs and 2 extra arcs.

Wong-Xiang Algorithm [Wong and Xiang, 1994; Xiang and Wong, 1994]

Wong-Xiang algorithm is an entropy based Markov network learning algorithm. In [Xiang and Wong, 1994], they proved that their algorithm can always learn a Markov network that is an I-map of the underlying model, and when the underlying model is a singly connected network, a minimum I-map is guaranteed to be constructed. Chu and Xiang have also developed a parallel version of the above algorithm [Chu and Xiang, 1997]. Using a pre-processed format of the ALARM network data with 10,000 cases, the running time can be reduced to 6 minutes on 12 processors from over 50 minutes on a single processor.

BENEDICT Algorithm [Acid and Campos, 1996b]

This is another entropy score based algorithm. As the Kutato algorithm, it requires node ordering and employs a heuristic search method. However, it uses a different entropy score to measure the closeness between a learned structure and the data. After getting a structure using heuristic search, the algorithm analyzes the conditional independencies implied in the structure by using the concept of d-separation (Section 2.2), and calculates the difference between these implied conditional independencies and the real conditional independencies of the data. Therefore, this algorithm actually

uses an entropy score that is the sum of the results of a group of CI tests. This algorithm has also been tested using the ALARM network data. Using 3000 cases of the ALARM data, it can generate a structure that has 4 missing arcs and 5 extra arcs compared to the true structure.

In [Sarkar and Murthy, 1996; Sarkar *et. al.*, 1996], an algorithm similar to BENEDICT is proposed.

CB Algorithm [Singh and Valtorta, 1995]

The CB algorithm is an early attempt to tackle the problem of search & score based algorithms, that node ordering must be provided by domain experts. Since dependency analysis based algorithms have the ability to orient edges, Singh and Valtorta developed a hybrid algorithm that employs both a dependency analysis method and a search & score method. CB first employs a modified version of the PC algorithm [Spirtes and Glymour, 1991] to find a node order and then uses a modified version of the K2 algorithm [Cooper and Herskovits, 1992] to learn the Bayesian network. Therefore, CB can avoid the requirement of node ordering from domain experts. Some more recent work on search & scoring based Bayesian network learning without node ordering is presented later this section. CB has been tested on ALARM network data. It can generate a structure with 2 missing arcs, 2 extra arcs and 2 wrongly oriented arcs.

Suzuki's Algorithm [Suzuki, 1996]

Suzuki's Bayesian belief network learning algorithm is based on the minimum description length (MDL) principle, which selects a rule that best compromises between simplicity and the fitness to data. The significance of this work is that, unlike most of the search & score based algorithms, this algorithm does not use heuristic search and it guarantees that the optimal structure is found. Since the search space is enormous, Suzuki developed a branch and bound technique, which calculates a lower-bound after an arc is added to the structure and determines if further search in this branch is necessary. This algorithm has also been tested using ALARM network data. The results show that on 100, 200, 500 and 1000 cases of ALARM data, the algorithm is more efficient and accurate than the K2 algorithm. However, it gets less efficient when the number of cases increases to several thousands.

Lam-Bacchus Algorithm [Lam and Bacchus, 1994]

The Lam-Bacchus algorithm is another minimal description length (MDL) scoring based algorithm. It uses the MDL principal in the same way as does the Suzuki algorithm. The significance of this work is that it does not need node ordering and it can orient the edges using a pure search & scoring method. This is quite different from the collider identification based edge orientation method used by many dependency analysis based algorithms and some hybrid algorithms. The authors also use the ALARM network to evaluate their algorithm. It can find a structure with three missing arcs and two wrongly oriented arcs compared to the real model.

Friedman-Goldszmidt Algorithm [Friedman and Goldszmidt, 1996]

This algorithm uses two different kinds of scores to learn Bayesian networks, i.e., MDL score and Bayesian score. The significance of this work is that it can learn the Bayesian networks and the local structures of the conditional probability distributions at the same time. (*Local structures* are used to replace the tables for representing conditional probability distributions associated with the nodes in a Bayesian network.) The authors show that their method has better performance than those methods that ignore the local structures. Like the Lam-Bacchus algorithm, this algorithm can orient edges using pure search & scoring method. When adding an edge, the algorithm chooses the direction of the edge that gives better score. The authors also use ALARM network data to evaluate their algorithm. However, they use the entropy distance to measure the differences between their results and the true model.

WKD (Wallace, Korb and Dai) Algorithm [Wallace *et. al.*, 1996]

The WKD algorithm discovers belief networks using the minimum message length (MML) method – a method similar to the minimum description length method used in several other algorithms described above. The authors use data sets from seven small domains to show that their results compare favorably with those of the PC algorithm [Spirtes and Glymour, 1991] even when the PC algorithm is supplied with node ordering and WKD is not.

6.3 Dependency analysis Based Methods

The Wermuth-Lauritzen Algorithm [Wermuth and Lauritzen, 1983]

This algorithm goes through the variables one by one according to the node ordering. For each node V_k to each node V_i such that $V_i < V_k$ apply a conditional independence test in the definition. If V_i and V_k are dependent, add $V_i \rightarrow V_k$ to the graph. This algorithm guarantees that a minimum I-map of a data set is generated. However, since it requires high order CI tests, it is only feasible for a large data set with very small number of variables to allow reliable CI tests. Therefore, this algorithm is highly impracticable.

Boundary DAG Algorithm [Pearl, 1988]

The boundary DAG algorithm is a simple way of building a Bayesian network given a node ordering and a joint probability function (or a large enough data set). With a proper search method, this algorithm can avoid most of the high order CI tests used in Wermuth-Lauritzen algorithm. However, since finding a Markov boundary of a node requires considering on every subset of the previous nodes in the ordering, this algorithm is exponential on the number of CI tests.

SRA (Srinivas, Russell and Agogino) Algorithm [Srinivas *et. al.*, 1990]

The SRA algorithm is a direct extension of boundary DAG algorithm. It loosens the requirement of complete node ordering by allowing partial node ordering and other forms of domain knowledge. When searching for a node to be added to the graph, it uses the partial ordering information and a heuristic search method to decide which node should be added. This algorithm requires an exponential number of CI tests.

Constructor Algorithm [Fung and Crawford, 1990]

This algorithm is similar to the DAG boundary algorithm in the sense that it learns the structure by finding the Markov boundary of each node. However, it tries to find a Markov network instead of a Bayesian network. So, it does not require node ordering. Although this method can avoid many unnecessary CI tests, it still has exponential performance cost. A difference between this algorithm and many other dependency analysis based algorithms is that, instead of using a fixed threshold in the CI tests, it uses a cross-validation technique to test different networks generated using different threshold settings, and select one best network. Therefore, it can avoid over-fitting to the data when the data set is not large enough.

SGS (Spirtes, Glymour and Scheines) Algorithm [Spirtes *et. al.*, 1990]

This algorithm is a Bayesian network learning algorithm that does not need node ordering. It can automatically orient edges of the learned structure using CI tests. This algorithm requires an exponential number of CI tests.

A variation of this algorithm is presented in [Verma and Pearl, 1992].

PC Algorithm [Spirtes and Glymour, 1991]

In 1991, Spirtes and Glymour enhanced their SGS algorithm so that it can be more efficient when constructing Bayesian networks from data sets that have sparse underlying models (models that do not have many edges). As for many other algorithms, the PC algorithm has also been tested using the ALARM network data. Using 10,000 cases of the ALARM data without node ordering, the algorithm can generate the network structure with 3 missing edges and 2 extra edges.

6.4 Other Algorithms

As a branch of search & scoring based approach, *model averaging* technique has been used by several researchers. They argue that sometimes the underlying model of a data set contains some uncertainty, that is, there is no single true structure that can represent the data perfectly. Therefore, instead of searching for a single best solution, some algorithms ([Buntine, 1994; Madigan and Raftery, 1994; Madigan *et. al.*, 1994]) return several networks and use the ‘average’ of these networks to perform belief propagation.

All the algorithms introduced in Section 6.2 and Section 6.3 assume that the data sets are causal sufficient – i.e., all the variables in the underlying models appear in the data sets. If some variables are not actually in the data sets, we call them *hidden variables* or *latent variables*. Some algorithms ([Spirtes *et. al.*, 1996; Spirtes *et. al.*, 1997; Verma and Pearl, 1990]) are trying to detect such variables.

There are also some algorithms that can handle data sets with missing values. A representative algorithm is described in [Ramoni and Sebastiani, 1996; Ramoni and Sebastiani, 1997]. The basic idea behind this algorithm is similar to that of the K2 algorithm. Ramoni and Sebastiani developed a method, called *Bound and Collapse*, to learn the parameters and structure from a data set.

7 Conclusions and Future Work

This chapter summarizes the work reported in this paper and proposes some future work directions.

7.1 Conclusion Remarks

This paper addresses the problem of learning Bayesian networks from data. Two information theoretic algorithms that are based on three-phase learning mechanism have been developed. The two algorithms are Algorithm A, which is used to learn Bayesian networks in a special case where node ordering is given, and Algorithm B, which is used to learn Bayesian networks in the general case. These two algorithms have already been implemented as a Bayesian network learning system, called Belief Network (BN) PowerConstructor. Using the PowerConstructor system, we have empirically evaluated our algorithms using two moderate complex real-world examples and one simple example. The results show that our algorithms are accurate and efficient.

The most significant advantage of our algorithms is that unlike all other practicable dependency analysis based algorithms, our algorithms can avoid exponential complexity on CI tests. Virtually, all dependency analysis based algorithms have to determine whether there should be an edge or not between a node and another node in the network, and $O(N^2)$ such decisions will give us a network structure. When node ordering is given, by using the three-phase mechanism, which allows some wrong decision to be made in Phase I and Phase II, Algorithm A manages to use one CI test for each such decision all the way till the correct structure is generated, whereas other algorithms use an exponential number of CI tests for each decision. Therefore, Algorithm A is of $O(N^2)$ on CI tests. It is proven correct when the underlying model of the data set is DAG-faithful. When node ordering is not given, although it is impossible to make each decision using one CI test by applying the three-phase mechanism alone, by also using quantitative CI tests, each of which can tell us not only whether a pair of nodes are dependent or not but also how close their relationship is, Algorithm B manages to use $O(N^2)$ CI tests for each decision. Therefore, Algorithm B is of $O(N^4)$ on CI tests. It is proven correct when the underlying model is monotone DAG-faithful.

These two polynomial algorithms have also been implemented in our BN PowerConstructor. Using the BN PowerConstructor system, we empirically evaluate our algorithms on three examples, the ALARM network, which has 37 attributes and 46 arcs, the Hailfinder network, which has 56 attributes and 66 arcs, and Chest-clinic network, which has 8 attributes and 8 arcs. The results show that our algorithms are capable of handling large real-world data sets since the running time is linear to the number of records in the data set and polynomial (roughly $O(N^2)$ for sparse networks) to the number of attributes in the data set. The results also show that our algorithms are quite reliable since the accuracy of the result does not deteriorate very fast when the sample size decreases.

The phenomenon that most running time is consumed by CI tests and the most running time of these CI tests is in turn consumed by database queries suggests a way to improve the efficiency of our algorithms – that is, to improve the efficiency of database queries. One method is to move the data set to a high performance database server. We believe this will speed up the Bayesian network learning from several to several hundred times depending on the speed of the database server. Another method is to use database engines that are specially designed for such queries, i.e., counting the number of records that satisfy certain criteria. This kind of database engines is demanded by many data mining

algorithms and the research in this area is undergoing. Parallelism is another way to improve the query efficiency. From the description of Algorithms A and B it is easy to see that Phase I, which uses over 60% of overall queries of each algorithm, can be easily parallelized.

7.2 Future Work

In theoretical aspect, we plan to work in the following directions.

1. Since there are generally two approaches to belief network learning, i.e., scoring & search based methods and dependency analysis based methods, and each approach has its own advantages, we plan to explore the possibility of combining the two approaches to improve the learning efficiency and accuracy and also to learn the models with hidden variables.
2. Algorithm B is correct for a subset of DAG-faithful models (we call them monotone DAG-faithful models). We conjecture that the assumption of monotone DAG-faithful is only slightly stronger than that of DAG-faithful and most DAG-faithful models are also monotone DAG-faithful. We plan to explore the properties of monotone DAG-faithful models and compare them with those of DAG-faithful models.
3. Our current work is focused on Bayesian network structure learning from data sets that have discrete values. We plan to expand it to Bayesian network parameter learning and allow missing values and continuous values in the data sets.

In practical aspect, we plan to develop a commercial version of the system and integrate it into large data mining, knowledge base and decision support systems. With this prototype at hand, we will implement our new research developments into it once the research reaches a practicable stage.

Bibliography

- [Acid and Campos, 1996a] Acid, S. and Campos, L.M., An algorithm for finding minimum d-Separating sets in belief networks, *Proceedings of the twelfth Conference of Uncertainty in Artificial Intelligence*, 1996.
- [Acid and Campos, 1996b] Acid, S. and Campos, L.M., BENEDICT: An algorithm for learning probabilistic belief networks, *Proceedings of the sixth International Conference IPMU'96*, 1996.
- [Agresti, 1990] Agresti, A., *Categorical data Analysis*, John Wiley & Sons, 1990.
- [Badsberg, 1992] Badsberg, J., Model search in contingency tables in CoCo, *Computational Statistics*, Dodge, Y and Wittaker, J. (eds), Heidelberg: Physica Verlag, page 251-256, 1992.
- [Beinlich *et al.*, 1989] Beinlich, I.A., Suermondt, H.J., Chavez, R.M. and Cooper, G.F., The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks. *Proceedings of the Second European Conference on Artificial Intelligence in Medicine* (pp.247-256), London, England, 1989.
- [Bouckaert, 1994] Bouckaert, R.R., Properties of Bayesian belief network learning algorithms. *Proceedings of the tenth conference on uncertainty in artificial intelligence*, Mantaras, R. and Poole, D. (Ed.), Morgan Kaufmann, 1994.
- [Buntine, 1994] Buntine, W., Operations for learning with graphical models, *Journal of Artificial Intelligence Research*, 2, page 159-225, 1994.
- [Buntine, 1996] Buntine, W., A guide to the literature on learning probabilistic networks from data. *IEEE Transactions on Knowledge and Data Engineering*, 8(2), 195-210, 1996.
- [Cheng *et al.*, 1997a] Cheng, J., Bell, D.A. and Liu, W., An algorithm for Bayesian belief network construction from data, *Proceedings of AI & STAT'97* (pp.83-90), Ft. Lauderdale, Florida, 1997.
- [Cheng *et al.*, 1997b] Cheng, J., Bell, D.A., and Liu, W., Learning belief networks from data: An information theory based approach, *Proceeding of the sixth ACM International Conference on Information and Knowledge Management*, 1997.

- [Cheng, 1998] Cheng, J., Learning Bayesian networks from data: An information theory based approach, *Doctoral Dissertation*, Faculty of Informatics, University of Ulster, U.K., 1998.
- [Chickering *et al.*, 1994] Chickering, D.M., Geiger, D., and Heckerman, D., Learning Bayesian Networks is NP-Hard, *Technical Report MSR-TR-94-17*, Microsoft Research, Microsoft Corporation, 1994
- [Chow and Liu, 1968] Chow, C.K. and Liu, C.N., Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14, 462-467, 1968.
- [Chrisman, 1996] Chrisman, L., A roadmap to Research on Bayesian networks and other decomposable probabilistic models, technical report, School of Computer Science, CMU, 1996.
- [Chu and Xiang, 1997] Chu, T. and Xiang, Y., Exploring parallelism in learning belief networks, *Proceedings of the thirteenth international conference on uncertainty in artificial intelligence*, 1997.
- [Cooper and Herskovits, 1992] Cooper, G.F., Herskovits, E., A Bayesian Method for the induction of probabilistic networks from data, *Machine Learning*, 9, 309-347, 1992.
- [Darroch *et al.*, 1980] Darroch, J.N., Lauritzen, S.L. and Speed, T.P. Markov fields and loglinear interaction models for contingency tables. *Ann. Stat.*, 8, 522-539, 1980.
- [Edwards, 1995] Edwards, D., *Introduction to Graphical Modelling*, Springer-Verlag, New York, 1995.
- [Friedman and Goldszmidt, 1996] Friedman, N. and Goldszmidt, M., Learning Bayesian networks with local structure, *Proceedings of the twelfth international conference on uncertainty in artificial intelligence*, 1996.
- [Fung and Crawford, 1990] Fung, R.M. and Crawford, S.L., Constructor: a system for the induction of probabilistic models, *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1990.
- [Geiger and Heckerman, 1990] Geiger, D. and Heckerman, D., Separable and transitive graphoids, *Proceedings of the sixth international conference on uncertainty in artificial intelligence*, 1990.
- [Geiger and Pearl, 1988] Geiger, D. and Pearl, J., Logical and algorithmic properties of conditional independence, Technical Report R-97, Cognitive Systems Laboratory, UCLA, 1988.
- [Glymour *et al.*, 1997] Glymour, C., Madigan, D., Pregibon, D. and Smyth, P., Statistical themes and lessons for data mining, *Data mining and knowledge discovery*, 1, 11-28, 1997.
- [Heckerman, 1988] Heckerman, D., An empirical comparison of three inference methods, *Uncertainty in Artificial Intelligence 4*, Shachter *et al.* (Ed.), page 283-302, North-Holland, 1988.
- [Heckerman, 1995] Heckerman, D., A tutorial on learning Bayesian networks, *Technical Report MSR-TR-95-06*, Microsoft Research, 1995.
- [Heckerman *et al.*, 1994] Heckerman, D., Geiger, D. and Chickering, D.M., Learning Bayesian networks: the combination of knowledge and statistical data, *Technical Report MSR-TR-94-09*, Microsoft Research, 1994. To appear, *Machine Learning Journal*.
- [Henrion, 1988] Henrion, M., Propagating uncertainty in Bayesian networks by probabilistic logic sampling, *Uncertainty in Artificial Intelligence 2*, page 149-163, North-Holland, 1988.
- [Henrion and Cooley, 1987] Henrion, M. and Cooley, D.R., An experimental comparison of knowledge engineering for expert systems and for decision analysis, *Proceedings AAAI-87 Sixth National Conference on Artificial Intelligence*, page 471-476, Morgan Kaufmann, 1987
- [Herskovits and Cooper, 1990] Herskovits, E. and Cooper, G., Kutato: An entropy-driven system for construction of probabilistic expert systems from databases, *Proceedings of the sixth international conference on uncertainty in artificial intelligence*, 1990.
- [Herskovits, 1991] Herskovits, E., Computer-based probabilistic network construction, *Doctoral dissertation*, Medical information sciences, Stanford University, Stanford, CA, 1991.

- [Hojsgaard *et al.*, 1994] Hojsgaard, S., Skjoth, F. and Thiesson, B., User's guide to BIOFROST, *Technical report*, Department of Mathematics and Computer Science, Aalborg, Denmark.
- [Krause, 1996] Krause, P., Learning probabilistic networks, *Technical Report*, Philips research laboratories, UK, 1996.
- [Lam and Bacchus, 1994] Lam, W. and Bacchus, F., Learning Bayesian belief networks: An approach based on the MDL principle, *Computational Intelligence*, Vol 10:4, 1994.
- [Lauritzen and Spiegelhalter, 1988] Lauritzen, S.L. and Spiegelhalter, D.J., Local computations with probabilities on graphical structures and their application to expert systems, *Royal Statistics Society B*, 50-2, 157-194, 1988.
- [Madigan and Raftery, 1994] Madigan, D. and Raftery, A.E., Model selection and accounting for model uncertainty in graphical models using Occam's window, *Journal of the American Statistical Association*, 89, page 1535-1546, 1994.
- [Madigan *et al.*, 1994] Madigan, D., Raftery, A.E., York, J.C., Bradshaw, J.M., and Almond R.G., Strategies for graphical model selection, *Selecting Models from Data: Artificial Intelligence and Statistics IV*, Cheeseman, P. and Oldford, R.W. (Ed.), Springer-Verlag, 1994.
- [Meek, 1995] Meek, C., Causal inference and causal explanation with background knowledge, *Proceedings of the eleventh international conference on uncertainty in artificial intelligence*, 1995.
- [Morjaia *et al.*, 1993] Morjaia, M.A., Rink, F.J., Smith, W.D., Klempner, J., Burns, C. and Stein, J., Commercialization of {EPRI}'s Generator Expert Monitoring System (GEMS), *Expert System Application for the Electric Power Industry*, EPRI, 1993.
- [Musick, 1988] Musick, C. R., Belief network induction, *doctoral dissertation*, UC Berkeley, 1988.
- [Neapolitan, 1990] Neapolitan, R.E. , *Probabilistic reasoning in expert systems: theory and algorithms*, John Wiley & Sons, 1990.
- [Pearl, 1988] Pearl, J., *Probabilistic reasoning in intelligent systems: networks of plausible inference*, Morgan Kaufmann, 1988.
- [Pearl and Wermuth, 1993] Pearl, J. and Wermuth, N., When can association graphs admit a causal interpretation? *Technical report*, 1993.
- [Ramoni and Sebastiani, 1996] Ramoni, M. and Sebastiani, P., Robust learning with missing data, *Technical report*, KMI-TR-28, 1996.
- [Ramoni and Sebastiani, 1997] Ramoni, M. and Sebastiani, P., Discovering Bayesian networks in incomplete databases, *Technical report KMI-TR-46*, Knowledge Media Institute, The Open University, March 1997.
- [Sarkar and Murthy, 1996] Sarkar, S. and Murthy, I., Constructing efficient belief network structures with expert provided information, *IEEE Transactions on knowledge and data engineering*, 8-1, 1996.
- [Sarkar *et al.*, 1996] Sarkar, S., Sriram, R.S., Joykuty, S. and Murthy, I., An information theoretic technique to design belief network based expert systems, *Decision support systems* 17, page 13-30, 1996.
- [Scheines *et al.*, 1994] Scheines, R., Spirtes, P., Glymour, C., and Meek, C., *TETRAD II: Tools for Discovery*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.
- [Singh and Valtorta, 1995] Singh, M. and Valtorta, M. Construction of Bayesian network structures from data: a brief survey and an efficient algorithm, *International Journal of Approximate Reasoning*, 12, 111-131, 1995.
- [Spiegelhalter *et al.*, 1993] Spiegelhalter, D., Dawid, P., Lauritzen, S., and Cowell, R., Bayesian analysis in expert systems, *Statistical Science*, 8:219-282, 1993.

- [Spirtes *et al.*, 1990] Spirtes, P., Glymour, C. and Scheines, R., *causality from probability, proceedings of Advanced Computing for the Social Sciences*, Williamsburgh, VA.
- [Spirtes *et al.*, 1991] Spirtes, P., Glymour, C. and Scheines, R., An algorithm for fast recovery of sparse causal graphs, *Social Science Computer Review*, 9, 62-72, 1991.
- [Spirtes *et al.*, 1996] Spirtes, P., Glymour, C. and Scheines, R., *Causation, Prediction, and Search* (Book), <http://hss.cmu.edu/html/departments/philosophy/TETRAD.BOOK/book.html>, 1996.
- [Spirtes *et al.*, 1997] Spirtes, P., Richardson, T., and Meek, C., Heuristic greedy search algorithms for latent variable models, *Proceedings of AI & STAT'97* (pp.481-488), Ft. Lauderdale, Florida, 1997.
- [Srinivas *et al.*, 1990] Srinivas, S. Russell, S. and Agogino, A., Automated construction of sparse Bayesian networks from unstructured probabilistic models and domain information, In Henrion, M., Shachter, R.D., Kanal, L.N. and Lemmer, J.F. (Eds.), *Uncertainty in artificial intelligence 5*, Amsterdam: North-Holland, 1990.
- [Suzuki, 1996] Suzuki, J., Learning Bayesian belief networks based on the MDL principle: An efficient algorithm using the branch and bound technique, *Proceedings of the international conference on machine learning*, Bari, Italy, 1996.
- [Thomas *et al.*, 1992] Thomas, A., Spiegelhalter, D.J., and Gilks, W.R., BUGS: A program to perform Bayesian inference using Gibbs sampling, *Bayesian Statistics 4*, Bernardo, J.M., Berger, J.O., Dawid, A.P. and Smith A.F. (eds), Oxford University Press, page 837-842, 1992.
- [Verma and Pearl, 1990] Verma, T.S. and Pearl, J., Equivalence and synthesis of causal models, *Proceedings of the sixth international conference on uncertainty in artificial intelligence*, 1990.
- [Verma and Pearl, 1992] Verma, T.S. and Pearl, J., An algorithm for deciding if a set of observed independencies has a causal explanation, *Proceedings of the eighth international conference on uncertainty in artificial intelligence*, 1992.
- [Wallace *et al.*, 1996] Wallace, C., Korb, K.B. and Dai, H., Causal discovery via MML, *Proceedings of the thirteenth international conference on machine learning (ICML'96)*, Morgan Kaufmann Publishers, San Francisco CA USA, pp. 516-524.
- [Wermuth and Lauritzen, 1983] Wermuth, N. and Lauritzen, S., Graphical and recursive models for contingency tables. *Biometrika*, 72, 537-552, 1983.
- [Whittaker, 1989] Whittaker, J., *Graphical Models in Applied Multivariate Statistics*, John Wiley & Sons, 1989.
- [Wong and Xiang, 1994] Wong, S.K.M. and Xiang, Y., Construction of a Markov network from data for probabilistic inference, *Third International Workshop on Rough Sets and Soft Computing*, pages 562-569, San Jose, CA, 1994.
- [Xiang and Wong, 1994] Xiang, Y. and Wong, S.K.M., Learning conditional independence relations from a probabilistic model, *Technical report*, University of Regina, 1994.