A Characterisation of Bayesian Network Structures and its Application to Learning

JAMES I.G. FORBES (jamesf@cs.monash.edu.au)
Computer Science Department
Monash University
Clayton, Victoria, 3168, AUSTRALIA

Abstract

We present an analysis of the minimal I-map relation between Bayesian network structures and dependency models. This includes a partial order characterisation of the structures, and the connection between the relation and the arc reversal operation. Two applications of this analysis are presented. The first is a simple condition for identifying equivalence between Bayesian network structures, and the second is an exact learning algorithm based on the partial order characterisation.

1 Introduction

A Bayesian network for a set of variables represents a joint probability distribution over those variables. It consists of two parts: a structure, which is a directed acyclic graph (DAG) representing the conditional independencies in the distribution, and a parameterisation corresponding to the structure. A network structure can have a number of parameterisation, thus representing a number of probability distributions. Two structures are *equivalent* [13] if they represent precisely the same distributions.

A dependency model over a set of variables is a collection of conditional independence relations between those variables. A probability distribution is a dependency model as is a DAG with the *d-separation* [7] criterion determining the independencies.

An important relationship between DAGs and dependency models is the *independency map*, or *I-map* relation. It determines whether a DAG encodes at least all of the dependencies in a dependency model[1]. A DAG is a *minimal I-map* if no edge can be removed from it whilst preserving the I-map condition.

In Section 3 we define a *minimal I-map class* to be the set of all minimal I-maps of a dependency model and derive a partial order characterisation of it. An important part of this is the relationship between the minimal I-map relation and the *edge reversal* [2] [9] operation on DAGs. There is a particular sequence of edge reversals between a DAG and its minimal I-maps, and we present an efficient algorithm for determining the sequence.

In Section 4 we discuss two applications of the characterisation from Section 3. The first of these is a simple test for equivalence of network structures, and the second is a learning algorithm based on the partial order of the minimal I-map class.

2 Preliminaries

Throughout this paper many comparisons are made between DAGs and their relationships to dependency models and probability distributions. It is assumed that all such entities are defined over the same set of variables $U = \{x_1, \ldots, x_n\}$.

A *topological sort* [3] of the variables in a DAG is a total ordering of the variables in which parents must precede their children.

The reversal of an edge $x \rightarrow y$ in a DAG $\mathcal{G}$ produces a DAG identical to $\mathcal{G}$, with $x \rightarrow y$ oriented as $y \rightarrow x$, and additional edges from the parents of x to y and from the parents of y to x if they did not already exist.

A dependency model [8] is a collection of conditional independencies, denoted by $I(X, Z, Y)$, meaning that the variables in X are conditionally independent of the variables in Y given the variables in Z. In this paper dependency models are assumed to be *graphoids* [8], that is they are closed under the symmetry, decomposition, weak union and contraction axioms.

[1] The independencies in the DAG are a subset of those in the dependency model.

[2] Shachter termed the operation arc reversal and here it is referred to as edge reversal.

[3] Sometimes simply referred to as an ordering of the DAG.

A *causal list*, also known as a *recursive basis* [5], over the set of variables U is the list of independence statements of the form, $I(x_i, \Pi_{x_i}, U_i \setminus \Pi_{x_i})$ where $U_i = \{x_1, \ldots, x_i\}$ and $\Pi_{x_i} \subseteq U_i$. A causal list can be used as an alternative representation of a DAG, as is quite often done in this paper. The ordering of the list is a topological sort of the corresponding DAG, and the variables in Π_{x_i} are the parents of x_i in the DAG. Verma and Pearl [12] have shown that independencies in the closure of a causal list under the graphoid axioms are precisely the same independencies identified by the d-separation criterion in the corresponding DAG. Therefore we can freely refer to a DAG, and its relationships with other DAGs and dependencies models, by a corresponding causal list.

A *boundary DAG/causal list* of a dependency model M is a minimal I-map of M constructed relative to a given ordering θ of the variables. Suppose $\theta =< x_1, \ldots, x_n >$ then the causal list $L =< \ldots, I(x_i, \Pi_{x_i}, U_i \setminus \Pi_{x_i}), \ldots >$ is constructed whereby each Π_{x_i} is the minimal subset of U_i so that the independency is in M. The set Π_{x_i} is called the *Markov boundary* of x_i relative to U_i. Verma and Pearl have shown that the DAG corresponding to L is a minimal I-map of M.

Verma and Pearl [13] have shown that two DAGs are equivalent if and only if they have the same skeleton and the same v-structures. Chickering [2] has shown that two DAGs are equivalent if and only if there exists a sequence of *covered edge* reversals from one to the other. An edge $x \rightarrow y$ is covered if x and y have the same parents, excluding x as y's parent. The equivalence of two DAGs $\mathcal{G}$ and $\mathcal{G}'$ is denoted by $\mathcal{G} \approx \mathcal{G}'$.

Much work has been done [11, 2, 6] on characterising equivalence classes by patterns using partially directed acyclic graphs (PDAGs). A PDAG can contain both directed and undirected edges and, for a DAG to be a member of an equivalence class represented by a PDAG then, it must have the same skeleton as the PDAG and the same directed edges. The directed edges are called *compelled* and the undirected are called *reversible*.

The following are definitions of a number of functions which are used in the remainder of the paper.

reverseEdge: *DAG* $\times$ *Edge* $\rightarrow$ *DAG*

reverseEdge$(\mathcal{G}, x \rightarrow y)$ reverses $x \rightarrow y$ in $\mathcal{G}$.

anEdge: *Causal List* $\times$ *Index* $\rightarrow$ {*true,false*}

anEdge(L, i) determines whether there is an edge between x_{i-1} and x_i in L. That is $x_{i-1} \in \Pi_{x_i}$

coveredEdge: *Causal List* $\times$ *Index* $\rightarrow$ {*true,false*}

coveredEdge(L, i) determines whether there is a covered edge from x_{i-1} to x_i in L. That is $\Pi_{x_i} = (\Pi_{x_{i-1}} \cup x_{i-1})$

reverseEdge: *Causal List* $\times$ *Index* $\rightarrow$ *Causal List*

reverseEdge(L, i) reverses the edge between x_{i-1} and x_i in L. There must be an edge between them.

swap: *Causal List* $\times$ *Index* $\rightarrow$ *Causal List*

swap(L, i) swaps x_{i-1} and x_i in the ordering of L. There must not be an edge between them.

exchange: *Causal List* $\times$ *Index* $\rightarrow$ *Causal List*

exchange(L, i) performs a swap of x_{i-1} and x_i if there is no edge between them otherwise, the edge is reversed.

3 Minimal I-Map Class

For a DAG to be a structure of a Bayesian network it must be a minimal I-map of the probability distribution represented by the network [7]. Therefore when considering Bayesian network structures we are only interested in the DAGs which are minimal I-maps of a probability distribution, or in general a dependency model. In this section we consider the set of all minimal I-maps of a dependency model and investigate some useful properties of it. The proofs of the theorems in this section are contained in [4].

Firstly we define a minimal I-map class of a dependency model.

Definition 1 *For a dependency model M, the* minimal I-map class *of M, denoted $\mathcal{M}(M)$, is the set of all DAGs which are minimal I-maps of M.*

The class $\mathcal{M}(M)$ contains all the boundary DAGs of M relative to all the possible orderings of the variables.

As with all sets of DAGs, $\mathcal{M}(M)$ can be partitioned into equivalence classes. These equivalence classes form a partial order with respect to the minimal I-map relation, when extended to equivalence classes. The relation can be extended to equivalence classes if it is consistent with in all classes. If a DAG $\mathcal{G}'$ is a minimal I-map of a DAG $\mathcal{G}$ then all the DAGs in the equivalence class of $\mathcal{G}'$ are minimal I-maps of all the DAGs in the equivalence class of $\mathcal{G}$. This is shown to be true in a minimal I-map class by the following theorem.

Theorem 1 *Let $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4 \in \mathcal{M}(M)$ for some dependency model M, and let $\mathcal{G}_2$ be a minimal I-map*

of $\mathcal{G}_1$. If $\mathcal{G}_3 \approx \mathcal{G}_1$ and $\mathcal{G}_4 \approx \mathcal{G}_2$ then $\mathcal{G}_4$ is a minimal I-map of $\mathcal{G}_3$.

The following theorem shows that the equivalence classes of a minimal I-map class form a partial order with respect to the minimal I-map relation.

Theorem 2 *For a dependency model M, the minimal I-map relation, extended to equivalence classes, defines a partial order on the equivalence classes of $\mathcal{M}(M)$.*

An example of such a partial order is given in Appendix A. In this example the dependency model is taken to be the DAG $\mathcal{G}_1$ and the table enumerates a DAG from each of the equivalence classes of $\mathcal{M}(\mathcal{G}_1)$. The diagram shows how the equivalence classes are arranged in the partial order.

The usefulness of this partial order characterisation of a minimal I-map class arises from the understanding of how the elements of the partial order are related. They are related not just by the minimal I-map relation, but also by the edge reversal operation. It is the case that if a DAG $\mathcal{G}'$ is a minimal I-map of a DAG $\mathcal{G}$ then there exists a sequence of edge reversals from $\mathcal{G}'$ to $\mathcal{G}$.

The sequence of edge reversals is identified by the *ForwardExchange* algorithm. If θ and θ' are topological sorts of the variables in $\mathcal{G}$ and $\mathcal{G}'$ respectively, then the *ForwardExchange* algorithm performs a sequence of edge reversals from $\mathcal{G}$ until the resulting DAG has the same topological sort as $\mathcal{G}'$, at which point it is $\mathcal{G}'$.

Function *ForwardExchange*:
$DAG \times Ordering \rightarrow DAG$

ForwardExchange$(\mathcal{G}, \theta)$

Performs a sequence of edge reversals from $\mathcal{G}$ so that the resulting DAG has θ as a topological sort of its variables.

1. Let $< x_1, \ldots, x_n >$ denote the ordering θ.
   Let θ' be a topological sort of the variables in $\mathcal{G}$ which *best matches* [a] θ.
   Let L be a causal list of $\mathcal{G}$ with θ' as its ordering.
2. **for** $i = n$ **down to** 2 **do**
   Let j be the position of x_i in L.
   for $k = j + 1$ **to** i **do**
   $L = exchange(L, k)$
3. **return** the DAG corresponding to L.

[a] This is important for ensuring the correctness of the algorithm. For a discussion, beyond the scope of this paper, see [4].

The correctness of the *ForwardExchange* algorithm is shown by the following theorem.

Theorem 3 *Let $\mathcal{G}$ and $\mathcal{G}'$ be DAGs and let θ' be a topological sort of the variables in $\mathcal{G}'$. $\mathcal{G}'$ is a minimal I-map of $\mathcal{G}$ if and only if $\mathcal{G}' =$* ForwardExchange$(\mathcal{G}, \theta')$.

To see how the algorithm works consider a DAG which has $< x_1, \ldots, x_n >$ as a topological sort of its variables and assume we want the minimal I-map of it which has the reverse ordering $< x_n, \ldots, x_1 >$ as a topological sort of its variables. The *ForwardExchange* algorithm works by firstly considering the variable in the last position of the required ordering, in this case x_1, and continually exchanging it with adjacent variables until it is last in the ordering $< x_2, \ldots, x_n, x_1 >$. The same process occurs for the second last variable, x_2, and so on until the required ordering is reached.

We can also see how the algorithm works by considering the example in Appendix A. The DAG $\mathcal{G}_5$ is a minimal I-map of $\mathcal{G}_1$ and the sequence of exchanges performed by the *ForwardExchange* algorithm are as follows.

DAG	θ	operation
$\mathcal{G}_1$	$<abcd>$	$reverseEdge(\mathcal{G}_1, b \rightarrow c)$
$\mathcal{G}_3$	$<acbd>$	$reverseEdge(\mathcal{G}_3, c \rightarrow d)$
$\mathcal{G}_6$	$<adcb>$	

4 Applications

In this section we look at two applications of the minimal I-map characterisation of Bayesian network structures. The first of these is a simple test for equivalence of structures and the second is a Bayesian network structure learning algorithm. The proofs of the theorems in this section are contained in [3].

4.1 Identifying Equivalence

The *ForwardExchange* algorithm identifies the sequence of edge reversals from a DAG to any of its minimal I-maps. If two DAGs are equivalent then they are a perfect map of each other which also means they are minimal I-maps of each other. Therefore it must be the case that the *ForwardExchange* algorithm will identify the sequence of edge reversals from one of the DAGs to the other, and vice versa. This gives us a test for identifying equivalence using the *ForwardExchange* algorithm.

Let $\mathcal{G}$ and $\mathcal{G}'$ be DAGs and θ and θ' be topological sorts of the variables in $\mathcal{G}$ and $\mathcal{G}'$ respectively. They are equivalent if and only if

$$\mathcal{G}' = \mathit{ForwardExchange}(\mathcal{G}, \theta'), \text{ and}$$
$$\mathcal{G} = \mathit{ForwardExchange}(\mathcal{G}', \theta).$$

We can remove the need to perform one of the *ForwardExchange* operations by making use of the result by Chickering [2]. That is two DAGs are equivalent if and only if there is a sequence of edge reversals between them in which each edge reversed is covered. This means that when performing a *ForwardExchange* we need to ensure that each edge reversed is covered. This is the case in the *ForwardCoveredExchange* algorithm which is a modification of the *ForwardExchange* algorithm.

Function *ForwardCoveredExchange*:
$DAG \times Ordering \rightarrow DAG$

ForwardCoveredExchange$(\mathcal{G}, \theta)$

Performs a sequence of covered edge reversals, if possible, from $\mathcal{G}$ so that the resulting DAG has θ as a topological sort of its variables.

1. Let $< x_1, \ldots, x_n >$ denote the ordering θ. Let L be a causal list of $\mathcal{G}$.
2. **for** $i = n$ **down to** 2 **do**
   Let j be the position of x_i in L.
   for $k = j + 1$ **to** i **do**
   if *anEdge*(L, k) **and**
   not *coveredEdge*(L, k) **then**
   return the DAG corresponding to L
   else
   $L =$ *exchange*(L, k)
3. **return** the DAG corresponding to L

The *ForwardCoveredExchange* algorithm performs the same sequence of edge reversals as the *ForwardExchange* algorithm, however it stops prematurely if an uncovered edge is to be reversed, thus ensuring the resulting DAG is equivalent to the given DAG.

If $\mathcal{G}'$ is equivalent to $\mathcal{G}$ then $\mathcal{G}'$ is the minimal I-map of $\mathcal{G}$ with θ' as a topological sort of its variables. Since the *ForwardExchange* algorithm will produce the minimal I-map of $\mathcal{G}$ with topological sort θ', it will produce $\mathcal{G}'$ by a sequence of edge reversals. The edges which are reversed are covered and therefore the *ForwardCoveredExchange* algorithm will also produce $\mathcal{G}'$.

This then gives us a simpler test for equivalence. $\mathcal{G}'$ is equivalent to $\mathcal{G}$ if and only if

$$\mathcal{G}' = \mathit{ForwardCoveredExchange}(\mathcal{G}, \theta').$$

Equally well they are equivalent if and only if

$$\mathcal{G} = \mathit{ForwardCoveredExchange}(\mathcal{G}', \theta).$$

It is a necessary condition of equivalence that only the reversible edges in a DAG have opposite orientation in an equivalent DAG. The *ForwardCoveredExchange* algorithm provides a sufficient condition for identifying which of the reversible edges can be as such. It also identifies the order in which they are reversed and is similar in method to the **Find-Edge** and **Build-Sequences** algorithms of Chickering.

4.2 An Exact Learning Algorithm

In this section we present an exact Bayesian network structure learning algorithm with similarities to that of Bouckaert [1]. The main similarity is the common approach of reordering the variables to find the minimal I-map with the least number of edges. The definition of each function comprising the algorithm are contained in Appendix B.

To describe the algorithm we consider the dependency model M and its minimal I-map class $\mathcal{M}(M)$. This class constitutes our search space and as the previous section revealed, its equivalence classes have the characteristic of being a partial order with respect to the minimal I-map relation.

It is the aim of exact learning algorithms, such as the IC-algorithm of Verma and Pearl [13], the PC-algorithm of Spirtes and Glymour [10] and the causal reordering algorithm of Bouckaert, to discover a DAG $\mathcal{G}_m$ which is the minimal I-map of M with the fewest edges. In terms of the partial order characterisation, $\mathcal{G}_m$ is the DAG such that its equivalence class is a maximal element of $\mathcal{M}(M)$[4].

A boundary DAG $\mathcal{G}_s$ is constructed from M relative to an arbitrary or given ordering of the variables, and is the starting point of the algorithm. The *ForwardExchange* algorithm identifies the sequence of edge reversals from a maximal element $\mathcal{G}_m$ to $\mathcal{G}_s$ thus providing a means of making downward transitions through the partial order. The learning algorithm performs the opposite, un-doing the edge reversals and thus making upward transitions.

We can see an example of this from the minimal I-map class in Appendix A. The class has only one maximal element which is the equivalence class of

[4] It may be possible that DAGs from different maximal elements have different numbers of edges. The extent to which they are different has not been investigated and is assumed to be minor.

$\mathcal{G}_1$. Therefore $\mathcal{G}_1$ is the goal of the algorithm. Starting with $< adcb >$ as our ordering, and $\mathcal{G}_5$ the corresponding boundary DAG, the algorithm will make an upward transition to $\mathcal{G}_3$ and from there an upward transition to $\mathcal{G}_1$.

Firstly we need to understand how to make upward transitions in the partial order. This is done by performing the opposite of a downward transition, which occurs from the reversal of an uncovered edge. The opposite is the *unreversal* of a covered edge. When an uncovered edge $x \rightarrow y$ is reversed in a DAG $\mathcal{G}$, the edge $y \rightarrow x$ in the resulting DAG $\mathcal{G}'$ has additional edges from the parents of x to y and vice versa. The unreversal of the edge $y \rightarrow x$ in $\mathcal{G}'$ removes the additional edges resulting in $\mathcal{G}$.

Knowing which edges to remove in the edge unreversal operation requires the dependency model M. Assuming $\mathcal{G}$ and $\mathcal{G}'$ are members of $\mathcal{M}(M)$ then $\mathcal{G}$ is the boundary DAG of M relative to $< \ldots, x, y, \ldots >$ and $\mathcal{G}'$ is the boundary DAG relative to $< \ldots, y, x, \ldots >$. Therefore unreversing the covered edge $y \rightarrow x$ in $\mathcal{G}'$ results in the boundary DAG[5] of M relative to $< \ldots, x, y, \ldots >$. This gives us the following additional operation for DAGs and causal lists similar to edge reversal.

unreverseEdge: *DAG* × *Dependency Model* × *Edge* → *(DAG,{true,false})*

unreverseEdge: *Causal List* × *Dependency Model* × *Index* → *(Causal List,{true,false})*

The *true* or *false* result depends on whether the edge in the resulting DAG/causal list is covered or not. The result is *true* if it is uncovered and *false* if it is covered, which identifies whether or not edges were removed and an upward transition made.

The *unreverseEdge* function is similar to the *swap* operator of Bouckaert. Bouckaert showed that swapping adjacent variables x and y will only bring about a reduction if the edge between them is covered. This is why the unreverse edge function is only applied to covered edges.

4.2.1 FindMaximal Function

The main part of the learning algorithm is the *FindMaximal* function.

The *ForwardExchange* algorithm transforms the ordering of $\mathcal{G}_m$ into that of $\mathcal{G}_s$ by firstly positioning the last variable of $\mathcal{G}_s$, then the second last and so on until we have the ordering of $\mathcal{G}_s$. Each variable is positioned by exchanging it with succeeding variables in the ordering.

[5] In practice the entire boundary DAG does not need to be determined, just the new parent sets of x and y.

The *FindMaximal* function works in the opposite way. The first variable in the ordering of $\mathcal{G}_s$, then the second and so on, are repositioned to where they are in the ordering of $\mathcal{G}_m$[6].

During the operation of the *ForwardExchange* algorithm each variable x_i is positioned by exchanging it with succeeding variables. Any edges between x_i and these variables are reversed and become covered. For each variable x_i the *FindMaximal* function unreverses the edges and recovers the causal list prior to the positioning of x_i by the *ForwardExchange* algorithm.

The first and second variables in the ordering can be ignored because the first has no preceding variables, and if there exists an edge between the first and the second then unreversing it will not bring about a transition.

The correctness of the *FindMaximal* function is show by the following Theorem.

Theorem 4 *Let M be a dependency model and θ an ordering of the variables. If $\mathcal{G}$ = FindMaximal(M, θ) then the equivalence class of $\mathcal{G}$ is a maximal element of $\mathcal{M}(M)$.*

4.2.2 UnreverseFrom Function

For the variable x_i we denote the variables, which participate in an edge reversal when x_i is positioned by the *ForwardExchange* algorithm, by $\{x_{f_1}, \ldots, x_{f_k}\}$. The result of the edge reversals is that $\{x_{f_1}, \ldots, x_{f_k}, x_i\}$ becomes a clique.

Bouckaert defines the notion of a *restriction* between two variables as the same as saying the edge between them is not reversible. He also defines the *free* variables of a clique as those which are not mutually restricted. The reduction of a clique is achieved by reordering the free variables in it.

All the variables in the clique $\{x_{f_1}, \ldots, x_{f_k}, x_i\}$ are free and a reduction is achieved by unreversing the edges in the clique.

Bouckaert describes an *unclique* operation which reduces a clique by testing the $|F_i|!$ possible orderings of the free variables F_i in it. This means his algorithm has a worst case complexity of $O(n!)$.

To reduce a clique, or find an independency between two variables in general, we need only consider all the subsets of the other variables, such is the case in the IC-algorithm, and not all the possible orderings of them. In terms of edge reversal and unreversal, the parent sets of the two variables depend only on

[6] Or of an equivalent DAG.

their predecessors and not the ordering of the predecessors.

To reduce the clique $\{x_{f_1}, \ldots, x_{f_k}, x_i\}$ we need to unreverse each of the edges $x_{f_j} \rightarrow x_i$ with each of the subsets of the other variables as predecessors. Each subset of the other variables preceding x_{f_j} and x_i is referred to as a *context* and to reduce the clique each edge needs to be unreversed in each context.

The first step of the function groups the free parents, $\{x_{f_1}, \ldots, x_{f_k}\}$, of x_i so that they are adjacent in the ordering of L. The resulting indices, f_s and f_e represent the start and end positions of the free variables in L.

The second step unreverses each of the edges $x_{f_j} \rightarrow x_i$ in each of the contexts of the other variables. Firstly the $x_{f_k} \rightarrow x_i$ is unreversed in each context by the *Unwind* function. If no reduction of the clique occurs then x_{f_k} is moved to the front of the clique and the edge $x_{f_{k-1}} \rightarrow x_i$ is unreversed in each context. This continues until a reduction occurs or all the free variables are exhausted.

4.2.3 GroupParents Function

This function groups the parents of x_i so that if possible they are adjacent to each other and to x_i in the ordering. By doing so it is then possible to determine which are the free variables $\{x_{f_1}, \ldots, x_{f_k}\}$.

Bouckaert has shown that the free variables can be grouped as such and this is performed by steps 1 and 2. Any non-parent predecessors of x_i are positioned, if possible, to be successors of x_i in the ordering and this is done by reversing only covered edges. Therefore the resulting causal list is equivalent to the given one and we remain in the same position in the partial order.

The free variables are now adjacent to x_i and to each other in the ordering, $< \ldots, x_{f_1}, \ldots, x_{f_k}, x_i, \ldots >$, and step 3 determines whereabouts they start. Since they form a clique and the edges between them are reversible the edge $x_{f_k} \rightarrow x_i$ and each of the edges $x_{f_j} \rightarrow x_{f_{j+1}}$ are covered.

4.2.4 Unwind Function

This function unreverses the edge $x_{f_k} \rightarrow x_i$ in each context of the other free variables in an attempt to reduce the clique $\{x_{f_1}, \ldots, x_{f_k}, x_i\}$.

The function is recursive in nature and this is used to generate each context. By recursively including and excluding each of the other free variables from the context we are able to generate all 2^{k-1} of them.

Step 1 determines whether there are any free variables and is the terminating condition for the recursion. Step 2 unreverses the edge in the current context and the remaining steps produce each of the contexts.

Steps 3, 4 and 5 remove the free variable $x_{f_{k-1}}$, if it exists, from the predecessors of x_{f_k} and x_i and then recursively calls the function to tests all contexts which do not contain $x_{f_{k-1}}$.

If no reduction of the clique occurs then the edge is unreversed in all contexts containing $x_{f_{k-1}}$. This is performed by steps 6,7 and 8 where $x_{f_{k-1}}$ is moved to the start of the free variables and the function called with $x_{f_{k-1}}$ excluded from the free variables so that it remains a predecessor.

If a reduction of the clique occurs during the function then the *UnreverseFrom* function is used to continue reducing the clique. A reduction will occur when an edge between a free variable x_{f_k} and x_i is unreversed. This means we have successfully repositioned x_i up the ordering and the *UnreverseFrom* function is called to see if it can be repositioned further.

We have also determined at this point that x_i is a predecessor, in $\mathcal{G}_m$, of the free variables which now succeed it in the ordering. The *UnreverseFrom* function is used to reposition them to where they are in the ordering of $\mathcal{G}_m$.

5 Conclusion

We have investigated the minimal I-map relation and defined a minimal I-map class relative to a dependency model. It has been shown that the equivalence classes of this class form a partial order with respect to the minimal I-map relation.

We have also shown the connection between the minimal I-map relation and the edge reversal operation. The *ForwardExchange* algorithm presented was shown to be sufficient for identifying the sequence of edge reversals from a Bayesian network structure to any minimal I-map of it.

Two uses of this analysis of the minimal I-map relation were discussed. The first is a condition identifying equivalence between two structures. By using the property that two equivalent structures are minimal I-maps of each other, it must be true that the *ForwardExchange* algorithm will produce one of the structures from the other, and vice versa.

The second application was an exact learning algorithm based on the partial order characterisation. The aim of the algorithm is to discover a structure at the top of the partial order. We have shown how to

make upward transitions in the partial order by performing the edge unreversal operation. Using this operation we presented an algorithm which makes upward transitions from anywhere in the partial order to a maximal element.

References

[1] R.R. Bouckaert. Optimizing causal orderings for generating DAGs from data. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 9–16. Morgan Kaufmann, 1992.

[2] D.M. Chickering. A transformational characterization of equivalent Bayesian network structures. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 87–98. Morgan Kaufmann, 1995.

[3] J.I.G. Forbes. Applications of the minimal I-map characterisation of Bayesian network structures. Technical Report TR 96/279, Department of Computer Science, Monash University, Australia, 1996.

[4] J.I.G. Forbes. Characterisations of Bayesian network structures. Technical Report TR 96/262, Department of Computer Science, Monash University, Australia, 1996.

[5] D. Geiger, T.S. Verma, and J. Pearl. Identifying independence in Bayesian networks. *Networks*, pages 507–534, 1990.

[6] C. Meek. Causal inference and causal explanantion with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 1995.

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

[8] J. Pearl and T. Verma. The logic of representing dependencies by directed acyclic graphs. In *Proceedings of the AAAI*, pages 374–379, 1987.

[9] R.D. Shachter. An ordered examination of influence diagrams. *Networks*, 20:535–563, 1990.

[10] P. Spirtes and C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9(1):62–72, 1991.

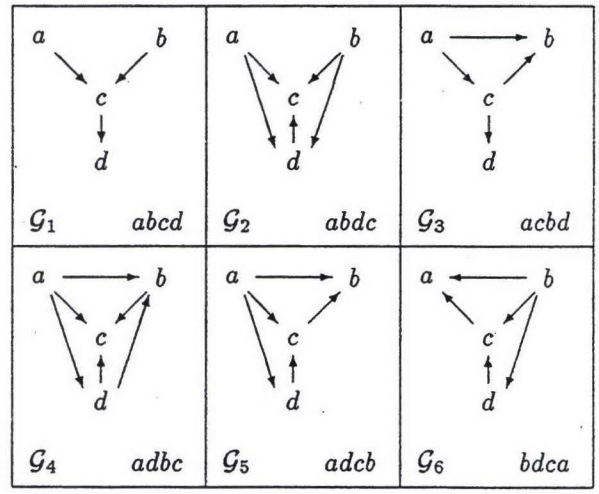
[11] T. Verma and J. Pearl. An algorithm for deciding if a set of observed independencies has a causal explanantion. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 1992.

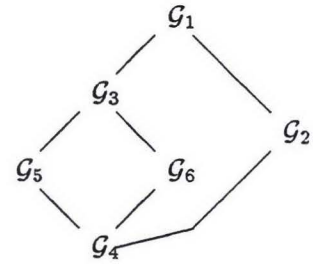
[12] T.S. Verma and J. Pearl. Causal networks: Semantics and expressiveness. In *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence*, pages 136–147, 1988.

[13] T.S. Verma and J. Pearl. Equivalence and synthesis of causal models. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 220–227, 1990.

Appendix A

Using $\mathcal{G}_1$ as a dependency model, the minimal I-map class $\mathcal{M}(\mathcal{G}_1)$ is partitioned into six equivalence classes. The table contains a DAG from each of the equivalence classes, and they are the boundary DAGs of $\mathcal{G}_1$ relative to the ordering of the variables in the lower right corner. The diagram below shows the arrangement of the equivalence classes in the partial order defined by the minimal I-map relation.





Appendix B

The following are the definitions of the functions comprising the exact Bayesian network structure learning algorithm.

Function *UnreverseFrom*:
Causal List × *Dependency Model* × *Index* → *Causal List*

UnreverseFrom(L, M, i)

1. $(L, f_s, f_e) =$ *GroupParents*(L, i)
2. **for** $i = f_s$ **to** f_e **do**
   (*L*,*reduced*) = *Unwind*(L, M, f_s, f_e)
   if *reduced* **then**
   return L
   else
   for $j = f_e$ **down to** f_s+1 **do**
   $L =$ *reverseEdge*(L, j)
3. **return** L

Function *GroupParents*:
Causal List × *Index* →
(Causal List,Index,Index)

GroupParents(L, i)

1. **while** $(i > 1)$ **and not** *anEdge*(L, i) **do**
   $L =$ *swap*(L, i)
   $i = i - 1$
2. $j = 1$
   while $j < i$ **do**
   if $x_j \notin \Pi_{x_i}$ **then**
   $k = j + 1$
   while $(k \leq i)$ **and**
   (**not** *anEdge*(L, k) **or**
   coveredEdge(L, k)) **do**
   $L =$ *exchange*(L, k)
   $k = k + 1$
   if $k > i$ **then**
   $i = i + 1$
   $j = j + 1$
3. $j = i$
   while *coveredEdge*(L, j) **do**
   $j = j - 1$
4. **return** $(L, j, i-1)$

Function *FindMaximal*:
Dependency Model × *Ordering* → *DAG*

FindMaximal(M, θ)

Returns a DAG $\mathcal{G}$ such that its equivalence class is a maximal element of $\mathcal{M}(M)$.

1. Let L be a boundary causal list of M relative to $\theta = < x_1, \ldots, x_n >$.
2. **for** $i = 3$ **to** n **do**
   $L =$ *UnreverseFrom*(L, M, i)
3. **return** the DAG corresponding to L.

Function *Unwind*:
Causal List × *Dependency Model* × *Index* × *Index* → *(Causal List,{true,false})*

Unwind(L, M, f_s, f_e)

1. **if** $f_s > f_e$ **then**
   return (*L*,*false*)
2. (*L*,*reduced*) = *unreverseEdge*(L, f_e+1)
   if *reduced* **then**
   $L =$ *UnreverseFrom*(L, M, f_e)
   $L =$ *UnreverseFrom*(L, M, f_e+1)
   return (*L*,*true*)
3. $L =$ *reverseEdge*(L, f_e+1)
   if $f_s = f_e$ **then**
   return (*L*,*false*)
4. $L =$ *reverseEdge*(L, f_e)
   (*L*,*reduced*) = *unreverseEdge*(L, M, f_e+1)
   if *reduced* **then**
   $L =$ *UnreverseFrom*(L, M, f_e)
   $L =$ *UnreverseFrom*(L, M, f_e+1)
   return (*L*,*true*)
5. (*L*,*reduced*) = *Unwind*(L, M, f_s, f_e-1)
   if *reduced* **then**
   $L =$ *UnreverseFrom*(L, M, f_e+1)
   return (*L*,*true*)
6. $L =$ *reverseEdge*(L, f_e+1)
   $L =$ *reverseEdge*(L, f_e)
   if $f_s \geq f_e-1$ **then**
   return (*L*,*false*)
7. **for** $i = f_e-1$ **down to** f_s+1 **do**
   $L =$ *reverseEdge*(L, i)
   (*L*,*reduced*) = *Unwind*(L, M, f_s+1, f_e)
   if *reduced* **then**
   return (*L*,*true*)
8. **for** $i = f_s+1$ **to** f_e-1 **do**
   $L =$ *reverseEdge*(L, i)
9. **return** (*L*,*false*)