LEARNING BELIEF NETWORKS IN PSEUDO INDEPENDENT DOMAINS

A THESIS
Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements
For the Degree of
Master of Science
In
Computer Science
University of Regina

By
Jun Hu
Regina, Saskatchewan
September, 1997

© Copyright 1997: Jun Hu
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-30479-5
Abstract

Belief networks (Bayesian networks, Markov networks, etc) are graphical representations for encoding uncertainty knowledge in artificial intelligence systems. They have been used in different applications including fault diagnosis, medical expert systems, and software debugging. Learning belief networks from data has been investigated by many researchers. Common learning algorithms employ a single link lookahead search. These algorithms cannot learn correctly when the domain model unknown to us is a pseudo independent (PI) model.

A PI model is a probabilistic domain model (PDM) where proper subsets of a set of collectively dependent variables display marginal independence. Earlier work on learning PI models has suggested a straightforward multi-link search algorithm. However, some PI models may escape the detection of such an algorithm. In this thesis, we study the relationship among PI submodels, and propose an improved algorithm that ensures the learning of all embedded PI submodels whose number of uncovered colored links are upper bounded by a predetermined parameter. We show that this improved learning capability only increases the complexity slightly beyond that of the previous algorithm.

The implementations are accomplished in DOS system and in Java language. An object-oriented model of belief networks is designed and used to implement the learning algorithm in Java. Experiments on real data and several constructed examples demonstrate the correctness and the effectiveness of the algorithm.

Keywords: Belief networks, probabilistic domain model, learning, search.
Acknowledgements

I first would like to express my grateful appreciation to my supervisor Dr. Y. Xiang for supporting me during my study, providing me the WEBWEAVR-II function library, and helping me conduct the research in my Master program.

I would like to thank Dr. Paul Gingrich for providing me the test data. Thanks are also given to Library and Information Services and Government Publications, University of Regina Library and the Social Sciences Computing Laboratory, Faculty of Arts, University of Regina for making these data available for academic use.

I acknowledge the financial support from the Natural Sciences and Engineering Research Council and the Institute for Robotics and Intelligent Systems in the Networks of Centres of Excellence Program of Canada. I would also like to acknowledge the Faculty of Graduate Studies and Research at the University of Regina for awarding me with Teaching Assistantship and Scholarship. All those who help me during this study are acknowledged.

_Dedicate to my lovely wife Yun Fan._
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Multi-link Lookahead Search</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>Review of Related works</td>
<td>23</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Learning from PI models</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Why Learning PI Models</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Early Works</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Learning Algorithm $RML$</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Properties of Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Complexity Analysis</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Implementations</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>DOS Implementation</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Why Java</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Java Implementation</td>
<td>44</td>
</tr>
<tr>
<td>4.3.1</td>
<td>System structure</td>
<td>44</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Object-oriented model</td>
<td>45</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Graphical user interface</td>
<td>49</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Experiments and Results</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>The ALARM Network</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>The PDM in Table 2.2</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>A Constructed Example: The Music-box Model</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>A Real Dataset: The 1993 GSS on Personal Risk</td>
<td>58</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>6.1</td>
<td>Contribution</td>
<td>61</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>62</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Another Experiment</td>
<td>68</td>
</tr>
<tr>
<td>Appendix B</td>
<td>PI Submodels that Share Nodes</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Tables

2.1 A full PI model ...................................... 14
2.2 A model with embedded PI submodels. ................. 15
3.1 The relation among i, steps per pass and passes in an i-link-only search. 38
5.1 A dataset generated from ALARM network ............... 53
5.2 The compressed frequency table of the ALARM network .... 53
5.3 Summary of learning the PDM in Table 2.2 ............... 55
5.4 Summary of learning the music-box model .................. 57
5.5 Variables in Harmful drinking model ...................... 58
5.6 States format for D15x ................................ 59
5.7 States format for D16x ................................ 59
A.1 Variables in Accident Prevention model. ................ 68
A.2 States format for A24x ................................ 69
A.3 States format for A25x ................................ 69
B.1 Example 17: A model $\mathcal{M}_\alpha$ with two PI submodels sharing nodes .... 72
B.2 Example 18: A model $\mathcal{M}_\beta$ with two PI submodels sharing nodes .... 72
B.3 Example 19: A model $\mathcal{M}_\gamma$ with two PI submodels sharing nodes .... 72
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Chordal graph and junction forest</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A Bayesian network: fire</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>A Markov network: fire</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Colored I-map of the model in Table 2.1.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Colored I-map of the model in Table 2.2.</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Colored I-maps of the four PI submodels in Table 2.2.</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Illustration of type 1 and type 2 links</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Illustration of two types of links in $L$.</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Applying the single link lookahead search to a PI model.</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Applying the straightforward multi-link search to a PI model.</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>The learning stages of algorithm $RML$.</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>The process of learning the model in Table 2.2.</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Illustration of covered colored links.</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Illustration of non-covered colored links.</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>Illustration of a PI submodel coverage DAG.</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>The execution chain of a 3-link search.</td>
<td>39</td>
</tr>
<tr>
<td>3.9</td>
<td>The execution forest of a 3-link search.</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>System structure of Java learning.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Object-oriented model of internal network</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>A run-time image of the graphical user interface</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Illustration of objects in the graphical user interface.</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>A run-time image of the dialogue box for editing environment variables.</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>The learned Markov network: ALARM</td>
<td>54</td>
</tr>
</tbody>
</table>
5.2 The specification of the music-box model. .................................. 55
5.3 Colored I-map of the music-box example. ............................... 56
5.4 The process of learning the music-box model. ........................... 56
5.5 The learned Markov network: *Harmful drinking* model .............. 60
5.6 The process of learning *Harmful drinking* model. ...................... 60
A.1 The process of learning *Accident Prevention* model .................. 70
A.2 The learned Markov network: *Accident Prevention* model .......... 70
Chapter 1

Introduction

1.1 Overview

Belief networks (Bayesian networks, Markov networks, etc) are increasingly being seen as a convenient high-level language for encoding uncertainty knowledge in artificial intelligence systems. They have been used in different applications including fault diagnosis, medical expert systems, and software debugging [16, 6, 18]. A belief network \((G, P)\) is a representation of the probabilistic dependence relationships among the variables in a problem domain, where \(G\) is a graph that represents the network structure and \(P\) is the probability distribution of the network. Each node in \(G\) corresponds to a variable in the problem domain. Each link corresponds to the dependence between the variables connected. The structure of a Bayesian network (BN) is a direct acyclic graph (DAG) and that of a Markov network is an undirect graph.

Belief networks are essentially used to answer probabilistic queries. The general problem of computing probabilities of interest from a joint probability distribution is called probabilistic inference. In the last twenty years, researchers have made significant progress in formalizing the theory of belief networks [30, 28], and in developing more efficient algorithms for probabilistic inference on belief networks [19, 22, 26, 29, 41].

Why is learning belief networks of particular interest? Although researchers have
made substantial advances in developing the theory and application of belief networks, the actual construction of these networks involved a tedious process of manual knowledge acquisition from domain experts. It is often a difficult, time-consuming task. For some domains with large size (number of variables), there may be few readily available experts. Methods are needed for augmenting the manual expert-based methods of knowledge acquisition for belief network construction.

As an alternative and supplement to manual knowledge acquisition, learning belief network from data has been researched actively by many researchers [20, 10, 24, 17, 3, 11, 27, 33, 38]. Once constructed, the belief networks can provide insight into the probabilistic dependences or independence among variables in the data and can be used for future probabilistic inference. Learning belief networks includes a number of complications: learning the structure, learning the parameters given a structure, learning hidden variables whose values are never present in the data, and learning from data where some tuples may contain missing values.

It has been shown that learning belief networks is NP-hard [2, 7]. Therefore, it is justified to use heuristic search in learning. Many algorithms developed use a scoring metric and a search procedure [20, 24, 32, 38]. The scoring metric evaluates the goodness-of-fit of a structure to the data, and the search procedure generates alternative structures and selects the best based on the evaluation. In these algorithms, a heuristic method with a single-link lookahead search is adopted in order to avoid the exponential complexity of exhaustive comparison of all possible networks. These algorithms rely on a single link lookahead search to identify local dependence among variables. In a single-link lookahead search, consecutive network structures differ by only one link.

Recently, the properties of models learnable by learning algorithms that use a single-link lookahead search have been investigated in [40]. The learnable models must satisfy two properties called \textit{Composition} and \textit{Strong transitivity}. The probabilistic domain models (PDMs) violating one of these two properties can not be learned correctly. If an incorrectly learned model is used for subsequent probabilistic inference, it will cause decision mistakes. Worse yet, the mistakes will be made without even knowing.
1.2 Motivation

It has been shown that pseudo-independent (PI) models are a class of PDMs that cannot be learned correctly by single-link lookahead search [43, 42]. The reason is that PI models violate a property called Composition[40] (to be defined in section 2.5). The pseudo-independent property of PI models requires multi-link lookahead search in order to detect the collective dependency [43]. As the computational complexity increases exponentially with the number of links to lookahead, a multi-link search must be performed cautiously. In order to manage the increased complexity, it is suggested [39] that the single link search should be performed first and then the number of links to lookahead should be increased one-by-one. A straightforward multi-link lookahead search was proposed in [43]. It will perform a single link lookahead search, then a double link lookahead search, and then a triple link lookahead search, etc. It turns out that some PI models will escape such a multi-link search. Therefore, Xiang [39] suggested to perform a single link lookahead search first, followed by a combination of double link lookahead and single link lookahead search, followed by a combination of triple, double and single link lookahead search, etc. However, it is unclear what is the most effective way to combine lookahead search of different number of links.

In this thesis, a more efficient method to manage the number of lookahead links is studied. An improved algorithm with multi-link lookahead search is proposed. After studying the relationships among PI submodels, the property of PI models learnable by this algorithm is analyzed. we show that this algorithm will correctly learn PI models in which every PI submodel satisfies a cardinality condition. The complexity of the algorithm and the experimental results are also presented. The experiments on real data provide solid evidence that PI models do exist in practice.

This learning algorithm has been implemented in two versions. Initially it is developed in DOS system based on a C-based function library. It is difficult to maintain and upgrade when the function library becomes larger and larger. Recently it is implemented in Java language. The Java implementation overcomes above drawbacks. It can run on different platforms (UNIX or Windows) without changing the source code. An object-oriented model of belief networks has been designed in this Java
version. This model can be used not only in developing this learning program, but also in developing other research tools on belief networks.

1.3 Organization

This thesis is organized as follows: Following the Introduction, Chapter 2 reviews the background knowledge about belief networks. The concepts of pseudo-independent (PI) models are presented with several examples. The minimum entropy approach for learning belief networks is discussed. The characteristics of models learnable by single link lookahead search are introduced. A multi-link lookahead search algorithm is also presented and analyzed. In Chapter 3, the need for learning PI models is discussed. After reviewing the early work on learning PI models, an improved multi-link search algorithm \textit{RML} is proposed. The property of algorithm is analyzed. The complexity analysis is also presented. Chapter 4 presents the Java implementation of the learning algorithm \textit{RML}. First, the drawbacks of its DOS implementation are analyzed. Next, the advantages of Java language are presented. For the Java learning program, the system structure and the object-oriented model are presented. The graphical user interface is also illustrated by several run-time images. Chapter 5 presents the detailed experimental results. Chapter 6 summarizes the contributions of this thesis and suggests future works.
Chapter 2

Background

This chapter introduces the background knowledge of belief network and reviews the related works. Section 2.1 describes the terminologies related to probability and graphs. Section 2.2 introduces Bayesian and Markov Networks. Section 2.3 introduces the definitions of pseudo-independent (PI) models. Two well known PI models are also described. The concept of minimum entropy approach is presented in Section 2.4. Section 2.5 describes the properties of models learnable by single link lookahead search. Since PI models violate one of these properties, they require multi-link lookahead search to discover the links missed in single-link search. The minimum entropy multi-link search algorithm is presented in Section 2.6. The last section, Section 2.7, reviews previous works related to this study.

2.1 Terminology

2.1.1 Probability related terminology

Let $N$ be a set of discrete variables $\{X_1, X_2, \ldots, X_n\}$ in a problem domain. Each variable is associated with finite number of possible values that it can take. We shall denote the possible values by consecutive integers 0, 1, 2, $\ldots$. A configuration or a tuple of $N' \subseteq N$ is an assignment of values to every variable in $N'$, e.g.,

$$(X_1 = 0, X_2 = 1, \ldots)$$
which we denote by
\[(x_{1,0}, x_{2,1}, \ldots).
\]

Let \(P(X_i)\) represents the probability function for \(X_i\) and \(P(x_i)\) denotes the probability value of \(P(X_i = x_i)\). The joint probability distribution (jpd) is \(P(N) = P(X_1, X_2, \ldots, X_N)\) and \(P(x_{1,0}, x_{2,1}, \ldots, x_{n,0})\) denotes the probability of a particular tuple of \(N\). A probabilistic domain model (PDM) \(\mathcal{M}\) over \(N\) determines the probability of every tuple of \(N'\) for each \(N' \subseteq N\).

For three disjoint subsets \(A, B\) and \(C\) in \(N\), \(A\) and \(B\) are conditionally independent given \(C\), denotes as \(I(A, C, B)_\mathcal{M}\), if
\[P(A|B, C) = P(A|C)\quad\text{whenever } P(B, C) > 0.\]

When \(C = \phi\), \(A\) and \(B\) are marginally independent. If each variables \(X\) in a subset \(A\) is marginally independent of \(A \setminus X\), then
\[P(A) = \prod_{X \in A} P(X).\]

We shall say that variables in \(A\) are marginally independent. A pair of variables \(X\) and \(Y\) are pairwise dependent if
\[P(X|Y) \neq P(X).\]

A set \(N\) of variables are collectively dependent if for each proper subset \(A \subset N\), there exists no proper subset \(C \subset N \setminus A\) such that
\[P(A|N \setminus A) = P(A|C).\]

A set \(N\) of variables are generally dependent if for any proper subset \(A\),
\[P(A|N \setminus A) \neq P(A).\]

### 2.1.2 Graph related terminology

Let \(G = (V, E)\) represents a graph where \(V\) denotes the set of vertices or nodes of the graph and \(E\) denotes the set of edges or links in the graph. Given a set
N of variables, we can associate a graph $G = (V, E)$ such that there is a one-to-one correspondence between variables in $N$ and nodes in $V$. Links in $E$ correspond to probabilistic dependences between the variables. Since we shall use nodes and variables interchangeably, we shall denote $G = (N, E)$ alternatively.

If $E$ contains only directed links, $G$ is a directed graph. For the two nodes connected by a directed link (an arrow), the node at the tail of the arrow is called the parent of the node at the head of the arrow. For each node $X$, its parent set is a set of nodes in which every node is a parent of node $X$. A directed graph without directed cycles is called a directed acyclic graph (DAG). A graph $G$ is an undirected graph if all the links in $E$ are undirected. A hypergraph $H_G$ is an undirected graph, which is converted from an undirected graph $G$ such that each node in $H_G$ is labeled by a set of nodes in $G$ and each link in $H_G$ is labeled by the intersection of the two nodes connected. A chord in an undirected graph is a link that connects two nonadjacent nodes. A graph is chordal if every cycle of length $> 3$ has a chord. A graph is complete if there are links between all pairs of nodes. A clique of a graph is a maximal set of nodes pairwise linked. A component of a graph is a maximal subgraph that is connected.

Let $G$ be a connected chordal graph. A junction tree (JT) $T$ of $G$ is a tree whose nodes are labeled by cliques of $G$ such that for each pair of nodes of $T$, their intersection is contained in every node on the unique path between them. A connected graph has a JT iff (if and only if) the graph is chordal [14]. In figure 2.1, $T_1$ is a JT of $G_1$. $T_1$ has four cliques, $\{a, c\}, \{a, b, d\}, \{b, d, c\}$ and $\{c, f\}$. The intersection of two adjacent cliques in $T$ is called the sepset of the two cliques. In figure 2.1, the sepset of cliques $\{a, e\}$ and $\{a, b, d\}$ is $\{a\}$. In general, $G$ may not be connected. A junction forest (JF) $F$ of $G$ is a set of JTs each of which is a JT of one component of $G$. In figure 2.1, $F_2$ is a JF of $G_2$ and $F_2$ consists of two JTs. $T_1$ is a trivial JF of $G_1$. 


node with converging arrows is in $Z$ or has a descendent in $Z$, and (2) every other node is outside $Z$ [30].

Given a probabilistic domain model (PDM) $\mathcal{M}$ over a set $N$ of variables and a DAG $G = (N, E)$, $G$ is an independence map (I-map) of $\mathcal{M}$ if for all disjoint subsets $X, Y, Z$ of nodes we have $\langle X \mid Z \mid Y \rangle_G \Rightarrow I(X, Z, Y)_\mathcal{M}$ [30]. That is, in an I-map, variables that are graphically separated are independent. Variables not graphically separated, however, are not necessarily dependent. A minimal I-map is an I-map in which no link can be deleted such that it is still an I-map.

From the chain rule of basic probability theory we know that the joint probability distribution $P(N)$ of a BN $(G, P(N))$ can be represented as a product:

$$P(x_1, \ldots, x_n) = P(x_1)P(x_2|x_1)\ldots P(x_n|x_1, \ldots, x_{n-1}).$$

Suppose the nodes are indexed by the order consistent with the parent-child relation in the DAG. As each node is conditionally independent with other nondecedent nodes given it's parent, we obtain

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\pi_i),$$

where $\pi_i$ is the parent set of $x_i$ and $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values for $X_1, \ldots, X_n$.

Figure 2.2 shows a Bayesian network named fire. This network is over six variables $\{\text{Tampering}, \text{Fire}, \text{Alarm}, \text{Smoke}, \text{Leaving}, \text{Report}\}$. Each variable could be denoted by its first character, and has two states true or false. We can see that $\pi_T = \phi$, $\pi_F = \phi$, $\pi_A = \{T, F\}$, $\pi_S = \{F\}$, $\pi_L = \{A\}$, and $\pi_R = \{L\}$. The joint probability distribution can be expressed by:


Therefore, only six small conditional probability tables are needed to compute the joint probability distribution. Since all six variables are binary, we only need 24 terms in total while the joint table on all six variables will have 64 terms.

The general problem of computing probabilities of interest from a (possibly implicit) joint probability distribution is called probabilistic inference. Suppose we observe an evidence “Leaving is true”, denote as $l$. We want to compute the probability
distribution of Fire given this evidence, we can have

\[ P(F|l) = \frac{P(F, l)}{P(l)} = \frac{\sum_{T, A, S, R} P(T, F, A, S, l, R)}{\sum_{T, F, A, S, R} P(T, F, A, S, l, R)}. \]

However, this approach is not feasible for problem domains with a large number of variables because the number of parameters required by an explicit description of the joint probability distribution is exponential on the total number of variables. The conditional independencies encoded in a Bayesian network can make this computation more efficient. For the same example, we have

\[ P(F|l) = \frac{P(F) \sum_T P(T) \sum_A P(A|T, F) P(l|A) \sum_S P(S|F) \sum_R P(R|l)}{\sum_T P(T) \sum_F P(F) \sum_A P(A|T, F) P(l|A) \sum_S P(S|F) \sum_R P(R|l)}. \]

The conditional independence produces a decomposition of the joint probability distribution that can be used to reduce the dimension of the computations.

### 2.2.2 Markov networks

Let \(X, Y\) and \(Z\) be three subsets of nodes in an undirected graph \(G\). We use \(< X|Z|Y >_G\) to mean that nodes in \(Z\) intercept all paths between nodes of \(X\) and
nodes of $Y$. An undirected graph $G$ is an independence map (I-map) of $\mathcal{M}$ over $N$ if there is an one-to-one correspondence between nodes of $G$ and variables in $N$ such that for all disjoint subsets $X, Y$ and $Z$ of $N$, we have

$$<X|Z|Y>_G \implies I(X, Z, Y)_\mathcal{M}.$$  

A graph $G$ is a minimal I-map of $\mathcal{M}$ if deleting any link of $G$ would destroy its I-mapness[30].

Let $G = (N, E)$ be a chordal graph. $F$ be a JF of $G$, and $\mathcal{M}$ be a PDM over $N$. Let $C$ be a clique of $F$ and $S$ be a sepset of $F$. Let $P_\mathcal{M}(C)$ and $P_\mathcal{M}(S)$ be the marginal distributions over $C$ and $S$, respectively, defined by $\mathcal{M}$. The jpd

$$P(v) = \frac{\prod_C P_\mathcal{M}(c))}{\prod_S P_\mathcal{M}(s)}$$

is called the projected distribution of $\mathcal{M}$ on $G$ (or on $F$), where $v$ is a configuration of $N$, $c$ is the projection of $v$ to $C$, and $s$ is the projection of $v$ to $S$.

The pair $(G, P)$ is a decomposable Markov network (DMN) obtained by projecting $\mathcal{M}$ to $G$, where $G$ is the structure of the DMN and $P$ is the distribution of the DMN. For simplicity, we shall call $(G, P)$ a DMN from $\mathcal{M}$.

![Figure 2.3: A Markov network: fire](image)

Figure 2.3 shows a decomposable Markov network named fire. It has six variables \{Tampering, Fire, Alarm, Smoke, Leaving, Report\}. Each variable could be
denoted by its first character, and has two states true or false. Figure 2.3 (a) is the structure of the Markov network; Figure 2.3 (b) are its corresponding junction tree and the local probability tables for all cliques in the junction tree.

Its joint probability distribution can be expressed by

\[ P(T, F, A, S, L, R) = \frac{P(cq_0)P(cq_1)P(cq_2)P(cq_3)}{P(S_0)P(S_1)P(S_2)}. \]

Similar to Bayesian network, we only need small probability tables for the cliques in a DMN to specify the joint probability distribution. For the Markov network fire, only four small probability tables are needed, in which there are 20 belief values in total, while a joint table of all six variables has 64 values. A DMN can also be used to perform probabilistic inference.

2.3 Pseudo-independent (PI) Models

A pseudo-independent (PI) model is a probabilistic domain model (PDM) where proper subsets of a set of collectively dependent variables display marginal independence. To make this thesis self-contained, we introduce the basic concepts on PI models from [39].

In general, PI models can be classified into three types. The most restrictive type is full PI models.

**Definition 1 (Full PI model)** A PDM over a set \( N \) (\(|N| \geq 3\)) of variables is a full PI model if the following two conditions hold:

(S1) For each \( X \in N \), variables in \( N \setminus \{X\} \) are marginally independent.

(S2) Variables in \( N \) are collectively dependent.

In a full PI model, every proper subset of variables are marginally independent. This is relaxed in the partial PI models. In a partial PI model, not every proper subset of variables are marginally independent.

**Definition 2 (Partial PI model)** A PDM over a set \( N \) (\(|N| \geq 3\)) of variables is a partial PI model if the following three conditions hold:
(S1') There exists a partition \( \{N_1, \ldots, N_k\} \) \((k \geq 2)\) of \( N \) such that variables in each subset \( N_i \) are generally dependent, and for each \( X \in N_i \) and each \( Y \in N_j \) \((i \neq j)\), \( X \) and \( Y \) are marginally independent.

(S2) Variables in \( N \) are collectively dependent.

In a PI model, it may be the case that not all variables in the domain are collectively dependent. An embedded PI submodel displays the same dependence pattern of the previous PI models but involves only a proper subset of domain variables.

**Definition 3 (Embedded PI submodel)** Let a PDM be over a set \( N \) of generally dependent variables. A proper subset \( N' \subset N \) \(||N'|| \geq 3\) of variables forms an embedded PI submodel if the following two conditions hold:

(S4) \( N' \) forms a partial PI model.

(S5) The partition \( \{N_1, \ldots, N_k\} \) of \( N' \) by S1' extends into \( N \). That is, there is a partition \( \{A_1, \ldots, A_k\} \) of \( N \) such that \( N_i \subseteq A_i, \) \((i = 1, \ldots, k)\), and for each \( X \in A_i \) and each \( Y \in A_j \) \((i \neq j)\), \( X \) and \( Y \) are marginally independent.

In general, a PI model can contain one or more PI submodels, and this embedding can occur recursively for any finite number of times.

Since variables in a PI submodel are collectively dependent, in a minimal I-map of the PDM, the variables in the submodel are completely connected. The marginal independence between subsets in the submodel is thus unrepresented. The undirected I-maps can be extended into colored I-maps [39]. The marginal independence between subsets are highlighted in a colored I-map by coloring the corresponding links.

**Definition 4** An undirected graph \( G \) is a colored I-map of a PDM \( M \) over \( N \) if (1) \( G \) is a minimal I-map of \( M \), and (2) for each PI submodel \( m \), links between each pair of nodes from distinct marginally independent subsets in \( m \) are colored. Other links are referred to as black.
Table 2.1: A full PI model

<table>
<thead>
<tr>
<th>(a, b, c)</th>
<th>P(a, b, c)</th>
<th>(a, b, c)</th>
<th>P(a, b, c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0)</td>
<td>0.1</td>
<td>(1, 0, 0)</td>
<td>0.14</td>
</tr>
<tr>
<td>(0, 0, 1)</td>
<td>0.02</td>
<td>(1, 0, 1)</td>
<td>0.14</td>
</tr>
<tr>
<td>(0, 1, 0)</td>
<td>0.08</td>
<td>(1, 1, 0)</td>
<td>0.28</td>
</tr>
<tr>
<td>(0, 1, 1)</td>
<td>0.1</td>
<td>(1, 1, 1)</td>
<td>0.14</td>
</tr>
</tbody>
</table>

A full PI model is shown in Table 2.1. This PDM has three binary variables \( \{a, b, c\} \) which form a full PI model. Each pair of variables is marginally independent, e.g.,

\[
P(a_0|b_0) = P(a_0) = 0.3, \quad P(a_0|c_0) = P(a_0) = 0.3, \quad P(b_0|c_0) = P(b_0) = 0.4.
\]

But the three variables are collectively dependent, e.g.,

\[
P(a_0|b_0, c_0) = 0.1/0.24.
\]

![Figure 2.4: Colored I-map of the model in Table 2.1.](image)

A partial PI model is shown in Table 2.2. This PDM has four variables, which are partitioned into three independent subsets. It contains four PI models over

\[
N_1 = \{a, c, d\}, \quad N_2 = \{a, b, c\}, \quad N_3 = \{b, c, d\}, \quad N_4 = \{a, b, c, d\}.
\]
Table 2.2: A model with embedded PI submodels.

<table>
<thead>
<tr>
<th>(d, a, b, c)</th>
<th>P(d, a, b, c)</th>
<th>(d, a, b, c)</th>
<th>P(d, a, b, c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0, 0)</td>
<td>0.02</td>
<td>(1, 0, 0, 0)</td>
<td>0.03</td>
</tr>
<tr>
<td>(0, 0, 0, 1)</td>
<td>0.02</td>
<td>(1, 0, 0, 1)</td>
<td>0.01</td>
</tr>
<tr>
<td>(0, 0, 1, 0)</td>
<td>0.06</td>
<td>(1, 0, 1, 0)</td>
<td>0.01</td>
</tr>
<tr>
<td>(0, 0, 1, 1)</td>
<td>0</td>
<td>(1, 0, 1, 1)</td>
<td>0.05</td>
</tr>
<tr>
<td>(0, 1, 0, 0)</td>
<td>0.1</td>
<td>(1, 1, 0, 0)</td>
<td>0.09</td>
</tr>
<tr>
<td>(0, 1, 0, 1)</td>
<td>0.06</td>
<td>(1, 1, 0, 1)</td>
<td>0.07</td>
</tr>
<tr>
<td>(0, 1, 1, 0)</td>
<td>0.14</td>
<td>(1, 1, 1, 0)</td>
<td>0.15</td>
</tr>
<tr>
<td>(0, 1, 1, 1)</td>
<td>0.1</td>
<td>(1, 1, 1, 1)</td>
<td>0.09</td>
</tr>
</tbody>
</table>

$N_1$, $N_2$ and $N_3$ are three embedded PI submodels. They are all embedded in $N_4$. $N_4$ itself is a partial PI model. Figure 2.5 shows the colored I-map of this model. The colored links are drawn as dotted.

Figure 2.5: Colored I-map of the model in Table 2.2.

For example, $N_1$ is a partial PI submodel with a marginally independent partition $\{\{a\}, \{c, d\}\}$ (S4). Since $\{a\}$ and $\{c, d\}$ are marginally independent, e.g.,

$$P(a_1, c_0) = P(a_1)P(c_0) = 0.48.$$ 

However, $c$ and $d$ are pairwise dependent. e.g.,

$$P(c_0, d_0) = 0.32 \neq P(c_0)P(d_0) = 0.3.$$
The three variables are collectively dependent, e.g.,

\[ P(a_1|c_0, d_0) = 0.75, \]

but

\[ P(a_1|c_0) = 0.8, \quad P(a_1|d_0) = 0.8, \quad P(a_1) = 0.8. \]

This partition extends into a marginally independent partition \( \{a\}, \{b, c, d\} \) (S5). Therefore, \( N_1 \) is embedded in \( N_4 \).

The colored I-maps of the four PI submodels in Table 2.2 are shown in Figure 2.6.

![Diagram](image)  

**Figure 2.6: Colored I-maps of the four PI submodels in Table 2.2.**

The well known *parity* problems and *modulus addition* problems can be shown to be special cases of PI models [39].

The parity problem can be described as follows: A problem domain consists of a set of marginally independent input variables \( \{X_1, \ldots, X_{n-1}\} \) and an output variable \( X_n \). Each input variable \( X_i \) \((1 \leq i \leq n-1)\) takes the value 0 or 1 with equal chance. The output variable \( X_n \) takes 0 or 1 such that the total number of 1's is even for even parity.

A modulus addition problem can be described as follows: A problem domain consists of a set of marginally independent and uniformly distributed input variables \( \{X_1, \ldots, X_{n-1}\} \) and an output variable \( X_n \). Each \( X_i \) \((i = 1, \ldots, n)\) has the domain \( \{0, 1, \ldots, D_i - 1\} \), where \( D_i \geq 2 \) such that for each \( i < n, D_i = k_i D_n \), in which \( k_i \) is a positive integer. \( X_n \) is the sum of \( X_1, \ldots, X_{n-1} \) modulo \( D_n \).
The PI models we discovered in practice Appendix A.

2.4 Minimum Entropy Approaches

This section briefly introduces the cor
nally presented in [43].

Given a PDM $M$ over a set $N$ of vari
that is an approximation of $M$. The ent
distribution $P$ over $X$ is

$$H(X) = - \sum_x$$

where $x$ is a tuple of $X$. The entropy of $X$

$$H(N) = \sum_C H$$

where $C$ is a clique of $G$ and $S$ is a sepset.

To measure the closeness of $(G, P)$ to
entropy [23],

$$K(P_M, P) = \sum_v P_M$$

where $P_M$ is the true jpd defined by $M$ at
minimizes $K(P_M, P)$ will be considered as
shown [38] that

$$K(P_M, P) = H(N)$$

where $H(N)$ is the entropy of $N$ defined by
minimizing $K(P_M, P)$ can be achieved by
minimum entropy approach.
2.5 Models Learnable by Single Link Lookahead Search

To make this thesis self-contained, we introduce models learnable by single link lookahead search. The detailed discussion can be found in [40].

In order to characterize models learnable by algorithms using single-link lookahead search, a generalized learning algorithm LIM was proposed in [39]. LIM can be viewed as a generalization of several commonly used algorithms for learning belief network structures [20, 24, 33, 10, 38, 17].

Suppose an algorithm LIM for learning I-maps is equipped with a test whether
\[ P(X|Y, Z) = P(X|Z) \] holds (equivalent to \( I(X; Y, Z) \) defined in subsection 2.1.1) for three disjoint subsets of variables \( X, Y \) and \( Z \). Clearly, if we allow such test to be performed for arbitrary \( X, Y \) and \( Z \), then LIM will be able to learn an I-map of any PDM. Unfortunately, the complexity of LIM will be exponential. We therefore restrict LIM such that the test is only performed based on the currently learned graph in the following manner:

LIM starts with an empty graph \( G \). It systematically selects a link \( \{x, y\} \) not contained in \( G \) such that one of the following two cases is true:

1. \( x \) and \( y \) are contained in different component of \( G \).

2. Every node (at least one) adjacent to both \( x \) and \( y \) is adjacent to every other such node, and these nodes intercept every path between \( x \) and \( y \).

We shall call the links that satisfy the above conditions type1 and type2 links respectively. In Figure 2.7, the missing link \( (b, c) \) is a type 1 link, and \( (a, d) \) and \( (d, g) \) are type 2 links. For a type 1 link, LIM tests if \( P(x|y) = P(x) \) (equivalent to \( I(x, \phi, y) \)). For a type 2 link, LIM tests if \( P(x|y, C) = P(x|C) \) (equivalent to \( I(x, C, y) \)), where \( C \) is the set of nodes adjacent to both \( x \) and \( y \). If the test is negative, then the link \( \{x, y\} \) is added to the current graph. LIM repeats the above until no type 1 or type 2 links can be added.

The following theorem shows that LIM actually returns a chordal graph and therefore learns a DMN. Since the imperfect data can not provide sufficient sampling, it will introduce additional dependencies which do not exist in underlying PDM. Even
the data is perfect, the weak dependencies in the underlying PDM will make the inference too complicated. Therefore we only choose strong dependencies and ignore too weak dependencies. Due to these two points, the learned DMN will be an approximation of the data-generating PDM. We do not require the structure of a DMN to be a minimal I-map of the PDM as required in some literature (e.g., in [30]). DMN are closely related to BNs but are simpler to study for the purpose of this analysis [25].

**Theorem 5** For any PDM, LIM returns a chordal graph on termination.

The class of PDMs learnable by LIM can be characterized by the following properties:

**Definition 6** Let $X, Y, Z, V$ and $W$ be any disjoint subset of variables.
- **Composition:** $I(X, Y, Z) \& I(X, Y, W) \implies I(X, Y, Z \cup W)$.
- **Strong Transitivity:** $I(X, Y \cup V, Z) \& I(Y, Z \cup V, W) \implies I(X, Y \cup V, Z \cup W)$.

The following theorem shows that the Composition axiom rules out PI models. The generally dependent is not a restriction of the learnable models. When the underlying PDM is not generally dependent, the result is applicable to each independent submodel.

**Theorem 7** Let $M$ be a generally dependent PDM over $N$ that satisfies Composition. Then $M$ is non-PI.
LIM will return an I-map as long as the underlying PDM satisfies Composition and Strong Transitivity. This is shown in Corollary 8. Note that the general dependence can now be removed.

**Corollary 8** Let $M$ be a PDM that satisfies Composition and Strong Transitivity. Let $G$ be a chordal graph returned by LIM. Then $G$ is an I-map of $M$.

The importance of Corollary 8 lies in the generality of LIM. It implies that a PDM satisfying Composition and Strong Transitivity is learnable by any algorithm, for learning BNs or DMNs, equipped with a single-link search and some scoring metric equivalent to a conditional independence test.

Can PDMs violating Composition be learned by LIM in general? Theorems 5 and 7 show PI models as PDMs that violate Composition and are unlearnable by any algorithm, for learning BNs or DMNs, equipped with a single-link search and some scoring metric equivalent to a conditional independence test. The PI models require multi-link lookahead search to discover the links missed in a single-link search due to models' violation of Composition.

### 2.6 Multi-link Lookahead Search

Suppose a PI submodel over $N' \subseteq \mathcal{N}$ is partitioned into $k$ marginally independent subsets. If we lookahead by multiple links at each search step such that $N'$ is completely connected by a set of new links and test

$$P(X|Y, N' \setminus \{X, Y\}) = P(X|N' \setminus \{X, Y\})$$

where $(X, Y)$ is one of the new links, we will get a negative answer. This prompts the completion of $N'$ in the learned graph. We shall call such strategy a *multi-link lookahead search*.

A multi-link lookahead search may be implemented using different scoring metrics. Our implementation using the cross-entropy scoring metric is based on the algorithm in [43]. Instead of testing the conditional independence directly, a test of whether new links decrease the Kullback-Leibler cross entropy is performed.
This algorithm was developed on the basis of three assumptions. The first assumption is that the database variables are discrete. The second is that no cases in the database have missing variables. The latest work on learning from database that have missing variables can be found in [8, 31, 35]. The third assumption, sparseness assumption, is based on the following observations: Let \( \eta \) be the size of the largest collectively dependent submodel in the problem domain. The higher the value of \( \eta \), the less likely that a submodel of size \( \eta \) exists in the problem domain.

A submodel of size \( \eta \) forms a clique of that size. The sparseness assumption allows us to lookahead a small number of links such that we will not miss many embedded PI models. Based on this assumption, the search is bounded with two parameters \( k \) and \( \eta \) specified by the user. The size of cliques is bounded by \( \eta \). The size of PI models is bounded, by \( k \leq m(m - 1)/2 \), to \( m \leq \eta \).

An additional requirement is that the data size (number of cases) should be large enough such that the probability distribution up to the order of \( \eta \) can be estimated reliably.

The pseudo code of this algorithm is shown below.

The function \( \text{lookahead}(i) \) performs a multi-link search which examines only \( i \geq 1 \) links at each step, that is, alternative structures that differ from the current structure by \( i \) links are evaluated. The \( i \) links that decrease the entropy maximally are selected. If the corresponding entropy decrement is significant enough, the \( i \) links will be adopted and the search continues until no more links can be learned. We refer to this search as an \( i \)-link-only search.

A threshold \( \delta h \) is used to differentiate between a strong dependence and a weak one (may be due to noise). A greedy search can thus be applied (line 4 through 9) to avoid adding unnecessary links and links due to weak dependence.

The condition that \( L \) is implied by a single clique \( C \) means that all links in \( L \) are contained in the subgraph induced by \( C \). This requirement helps to reduce the search space. For example, let's consider \( \text{lookahead}(2) \) which performs a double-link-only search. Figure 2.8 (1) is the current graph. Figure 2.8 (2) is an alternative graph by adding the dashed lines, in which \( L \) is not implied by a single clique. Since a PI submodel can not be learned by adding them (a PI submodel forms a single
Algorithm 2.6

Function BOOL lookahead( int i);
Input: i is the number of lookahead links.
A graph \( G = (N, E) \) as a globe variable.
Comment: \( \delta h \) is a threshold.

begin
1 \hspace{1em} \textit{modified} := \textit{false};
2 \hspace{1em} \textit{G}' := \textit{G};
3 \hspace{1em} \textbf{repeat}
4 \hspace{2em} \text{initialize the entropy decrement } dh' := 0;
5 \hspace{2em} \text{for each set } L \text{ of links } (|L| = i, L \cap E = \phi), \text{ do}
6 \hspace{3em} \text{if } G^* = (N, E \cup L) \text{ is chordal and } L \text{ is implied by a clique,}
7 \hspace{4em} \text{then compute the entropy decrement } dh^*;
8 \hspace{3em} \text{if } dh^* > dh', \text{ then } dh' := dh^*, \textit{G}' := G^*;
9 \hspace{2em} \text{else done := true;}
10 \hspace{1em} \textbf{until done = true;}
11 \hspace{1em} \textit{return} \hspace{0.5em} \textit{modified};
end

clique), this alternative structure will not be further tested. Figure 2.8 (3) is another alternative graph by adding the dotted lines in which \( L \) is implied by a single clique \( \{c, d, e\} \). This alternative structure will be further tested (\( dh' \) computed) since it may form a PI submodel.

We briefly discuss the worst case time complexity of the algorithm. Testing the chordality of \( G^* \) can be performed in \( O(|N|) \) time [14]. In computing \( dh^* \), we can compute \( F \) and \( F^* \) in \( O(\eta^2 \log \eta) \) time [43]. The distribution \( P \) on the \( 2\eta \) variables from the database can be extracted directly in \( O(n) \) time, where \( n \) is the number of cases in the database. It takes \( O(\eta 2^n) \) time to compute \( dh^* \) from \( P \) [43]. The computation of \( dh^* \) from the projected distribution can be performed in \( O(2^n) \) time. The complexity of each step is then \( O(|N| + n + \eta(\eta \log \eta + 2^n)) \). Since \( n \) is much larger than \( |N| \), the complexity of each step is \( O(n + \eta(\eta \log \eta + 2^n)) \).

The algorithm repeats for \( O(|N|^2) \) passes, and each pass has \( O(|N|^{2k}) \) search steps.
Figure 2.8: Illustration of two types of links in $L$.

Hence the complexity of the algorithm is $O(|V|^2(k+1)(n + \eta \log \eta + 2\eta))$.

### 2.7 Review of Related works

The earliest work in structure learning was the Chow and Liu algorithm [9] for learning tree from data. A polytree (singly connected network) is a belief network that contains at most one undirected path between any two nodes in the network. Rebane and Pearl [32] used the Chow-Liu algorithm as the basis for an algorithm that recover polytree-structured BN from a probability distribution. This algorithm assumes the availability of a conditional independence (CI) test that determines whether variables in a set $X$ are independent of variables in a set $Y$, given the variables in a set $Z$. However, many real world domain models cannot be represented adequately by a polytree-structured BN.

Several algorithms have been developed that use a CI test to recover a multiply connected belief network, which is a belief network containing at least one pair of nodes that have at least two undirected paths between them. Wermuth and Lauritzen [36] describe a method that takes as input an ordering on all model nodes and then applies a CI test to a distribution to construct a belief network that is a minimal I-map. Srinivas, Russell, and Agogino [34] allow the user to specify a weaker set of constraints on the ordering of nodes, and then use a heuristic algorithm to search for a belief network I-map (possibly nonminimal). Spirtes, Glymour, and Scheines [33] have developed an algorithm, called PC, to learn a BN by deleting links from a complete graph. PC does not require a node ordering. Herskovits and Cooper [20]
developed the Kutato algorithm that use the entropy scoring metric and a greedy search to learn a BN from data. This algorithm starts with an empty graph (no links) and adds on link at each pass during the search process until a halting condition is reached. They also proposed the K2 algorithm [10] which learns a BN based on the Bayesian method that selects a BN with the highest posterior probability given a database. Lam and Bacchus [24] applied a minimal description length (MDL) method to learn a BN that is evaluated as the best if it has the minimal sum of its own encoding length and the encoding length of the data given the BN. Heckerman, Geiger and Chickering [17] applied the Bayesian method to learning a BN by combining prior knowledge and statistical data. Their approach is based on the assumption of likelihood equivalence, which means that data should not help to discriminate network structures representing the same assertions of conditional dependence. A more extensive review of literature for learning belief networks can be found in [4].

Pearl [30] showed that directionality of BN makes it a richer language in expressing dependencies. In general, fewer numerical parameters are required to specify a BN than those required to specify a corresponding DMN. However, learning of DMNs is useful for several reasons [43].

One important application of BNs is to compute posterior probabilities. One efficient exact algorithm [22] for doing that in a sparse multiply connected network uses a DMN, in terms of its junction tree (JT), as the run time representation of a BN. It has been shown [37, 38] that computation of posterior probabilities of a BN can be performed using an extended relational database once the BN is converted into its corresponding DMN. This implies that once a probabilistic model is expressed in terms of a DMN, inference can be performed using standard relational DBMSs. Finally, as BNs and DMNs are so closely related, knowledge gained in learning one of them will benefit the learning of the other.

lookahead search algorithm for learning a DMN. This algorithm can learn DMNs in Pseudo-independent (PI) models. Recently, Hu and Xiang [21] developed an improved algorithm to learn DMN in recursively embedded PI submodels. These two algorithms will be discussed in detail in the next chapter.
Chapter 3

Learning from PI models

This chapter presents learning DMNs from PI models. Section 3.1 discusses why learning from PI models is needed. Section 3.2 introduces the early works on learning from PI models. Section 3.3 presents the improved learning algorithm for learning from PI models. The learning process is illustrated by a constructed example. In section 3.4, the colored link coverage relationship among PI submodels is discussed. Then the property of this algorithm is presented. The algorithm complexity is analyzed in Section 3.5.

3.1 Why Learning PI Models

PI models do exist in practice (see section 2.3 for parity and modulus addition problems, and see section 5.4 and Appendix A for PI models found in real data). Commonly used algorithms for learning belief networks rely on a single link lookahead search to identify local dependence among variables. These algorithms cannot learn PI models correctly. In subsequent inference, an incorrectly learned model will cause decision mistakes.

If the single link lookahead search is applied to learn from the PDM $\mathcal{M}$ shown in Table 2.2. The learning result of single link lookahead search is shown as $G_1$ in Figure 3.1. $G_1$ is not a correct I-map of $\mathcal{M}$. Since $\mathcal{M}$ is a PI model, its correct I-map is shown as $G_2$ in Figure 3.1. If we want to calculate the probability of $c = 1$ given
the observation $a = 0$, $b = 0$, and $d = 0$. The model of $G_1$ shows:

$$P(c_1|a_0, b_0, d_0) = P(c_1|d_0) = 0.36.$$  

The model of $G_2$ shows:

$$P(c_1|a_0, b_0, d_0) = P(a_0, b_0, c_1, d_0)/P(a_0, b_0, d_0) = 0.5.$$  

This difference in the posteriors may warrant quite different actions.

Figure 3.1: Applying the single link lookahead search to a PI model.

3.2 Early Works

The pseudo independence property of PI models requires multi-link lookahead search in learning. A multi-link search is more expensive than a single link search since $O(|N|^2)$ sets of links need to be tested before one set of links is adopted. Since the complexity increases exponentially with the number of links to lookahead, an multi-link search must be performed cautiously. Three strategies are proposed in [39] to manage the computational complexity: (1) performing single link search first, (2) increasing the number of links to search one-by-one, and (3) making learning inference-oriented.

A straightforward multi-link search is suggested in [43]. Such a search will perform a single link lookahead, followed by a double link lookahead, followed by a triple link
lookahead, etc. This method can learn many PI models. However, a PDM with recursively embedded PI submodels may escape the straightforward algorithm. For example, if we apply the straightforward multi-link search to the PDM in Table 2.2. The single link search will add the link \((d, c)\). The followed double link search will first discover the PI submodel over \(N_1\) and add links \((a, d)\) and \((a, c)\). It then discovers the PI submodel \(N_3\) and adds links \((b, d)\) and \((b, c)\). But the PI submodel over \(N_2\) will never be learned by the double link lookahead or lookahead with higher number of links, since only one link \((a, b)\) is unconnected. Consequently, the learned graph shown in Figure 3.2 (b) is not an I-map.

![Figure 3.2: Applying the straightforward multi-link search to a PI model.](image)

Realizing this deficiency of the straightforward multi-link search, an improved multi-link search algorithm was proposed in [39]. In addition to the incorporation of the above three strategies, the search is performed in the following manner: A single link lookahead is performed first, followed by a combination of double link lookahead and single link lookahead, followed by a combination of triple, double and single link lookahead, etc. We shall refer to such a systematic search that lookaheads by no more than \(i > 1\) links as an \(i\)-link search.

The algorithm proposed in [39], however, did not specify what is the most effective way to combine lookahead search of different number of links. This is the issue we address in this chapter.
3.3 Learning Algorithm RML

We start by asking the question why some PI models may escape the straightforward multi-link search. The previous example shows that the main reason is the recursive embedding of PI submodels. If a PI submodel $M_1$ is embedded in another PI model $M_2$, $M_1$ will be learned first. After that, if the number of unlearned links in $M_2$ is less than the current number of links to lookahead, $M_2$ will not be learned correctly in the later search steps. In order to learn $M_2$, backtracking to lower number of lookahead links is necessary. Hence the problem translates to a proper arrangement of backtracking during learning.

We propose a multi-link search algorithm (RML) which overcomes the deficiency of the straightforward multi-link search. The learning outcome is represented as DMN. The algorithm first learns the chordal structure. Once the chordal graph is obtained, the numerical probability distribution can be estimated from the data.

RML starts with an empty graph. It performs a single link search first. The first stage of the search now ends.

RML then performs a double-link-only search. If some links are learned during the double-link-only search, RML backtracks to perform another single link search. Afterwards, it performs double-link-only search again and backtracks if necessary as before. The combination of double-link-only and single link search will continue until no link is learned in a double-link-only search. We shall refer to this repeated combination of the double-link-only search and the single link search as a combined-double-link search. Now the second stage of the search ends.

Next, RML will perform a triple-link-only search. If some links are learned during the search, RML backtracks to repeat the previous two stages. Afterwards, it performs another triple-link-only search and backtracks if necessary as before. We shall refer to this repeated combination of the triple-link-only, double-link-only and single link search as a combined-triple-link search. Note that a combined-triple-link search can include several combined-double-link search. Now the third stage of the search ends.

RML continues with a combined-four-link search, followed by a combined-five-link search, etc., until a combined-$k$-link search, where $k > 1$ is a predetermined integer.
The pseudo-code of this algorithm is presented below.

**Algorithm RML**

Input: A dataset $D$ over a set $N$ of variables.

    a maximum number $k$ of lookahead links.

Return: The learned graph.

Comment: $\text{lookahead}(i)$ is the function for an $i$-link-only search (see algorithm 2.6).

begin

1. initialize a graph $G = (N, E = \emptyset)$;

2. for $j := 1$ to $k$ do

3.     $i := j$;

4.     while $i \leq j$ do

5.         $\text{modified} := \text{lookahead}(i)$;

6.         if ($i > 1$) AND ($\text{modified} = \text{true}$)

7.             then $i := 1$: {backtracking}

8.         else $i := i + 1$:

9.     return $G$ and halts.

end

In algorithm RML, the search stages are indexed by $j$ (line 2) and each iteration of the outer for loop corresponds to one stage. The first iteration has $i = j = 1$ (lines 2 and 3). The single link search $\text{lookahead}(1)$ (line 5) will be performed. The test in line 6 will fail and $i$ becomes 2 (line 8). This terminates the while loop as well as the first iteration of the for loop. It corresponds to the first stage of search.

The next iteration of for loop has $i = j = 2$. The double-link-only search $\text{lookahead}(2)$ will be performed. If some links have been added, the test in line 6 will succeed and $i$ becomes 1. This causes the execution of another single link search $\text{lookahead}(1)$. Afterwards, $i$ becomes 2 and another double-link-only search will be performed. If nothing has been added, $\text{modified}$ is false and $i$ becomes 3. This terminates the while loop and the second iteration of the for loop. It corresponds to the second stage of search.

The next iteration of for loop has $i = j = 3$. At first, the triple-link-only search $\text{lookahead}(3)$ will be performed. If some links have been added, the test in line 6
will succeed and i becomes 1. This causes the repetition of the previous two stages. This execution of lookahead(3) and repetition of stages 1 and 2 continues until an execution of lookahead(3) returns false. Afterwards, i becomes 4 and the while loop will be terminated. It will also terminate the third iteration of the for loop and end the third stage of search.

The function lookahead(i) performs an i-link-only search. It consists of multiple passes and each pass is composed of multiple steps. Each step tests one set of i links. Each pass learns one set of i links after testing all distinct and legal combinations, one at each search step, of i links.

Figure 3.3: The learning stages of algorithm RML.

The learning stages of algorithm RML can be shown as Figure 3.3. In each stage, if new links are learned after a i-link-only (i > 1) search, RML will backtracks to perform single link search. Otherwise the number of lookahead links will be increased.

Figure 3.4 shows the execution of RML in learning the PI model in Table 2.2 with the value of k set as k = 2. RML starts with a single link search (The first stage). After all links are examined, one set of links \( L_1 = \{(d, c)\} \) is learned. The learned graph is shown in Figure 3.4 (a). In the second stage, RML performs the double-link-only search first, which learns two sets of links \( L_2 = \{(d, a), (a, c)\}, \ L_3 = \{(d, b), (b, c)\} \). These links are contained in the PI submodels over \( N_2 \) and \( N_3 \). The
corresponding graph is shown in Figure 3.4 (b). Since some new links are added after the double-link-only search, RML backtracks to perform the single link search again. During this search one set of links \( L_4 = \{(a, b)\} \) is added and Figure 3.4 (c) is obtained. RML continues to perform another double-link-only search but no more links can be learned. The RML halts with a complete graph which is a correct I-map.

The model in Table 2.2 is a special PI model since the entire domain is a partial PI model. Hence the learning algorithm returns a complete graph. In general, the learning algorithm will not return a complete graph if a PDM is not a partial PI model over the entire domain variables. Only the variables covered by an embedded PI submodel will be completely connected by the learning algorithm.

### 3.4 Properties of Algorithm

Can RML learn any PI model correctly? Clearly the answer is no as RML only searches up to a predetermined number \( i \) of lookahead links. A PI submodel that contains more than \( i \) colored links may escape RML. Then what is the characteristics of the PI models that can be learned by RML? In this section we answer this question.

First, we introduce several terms.

**Definition 9 (Marginally independent subsets)** Let \( \mathcal{M} \) be a PDM over a set \( N \) of generally dependent variables. Two disjoint nonempty subsets \( N_1 \) and \( N_2 \) of \( N \) are marginally independent subsets if for each \( X \in N_1 \) and \( Y \in N_2 \), \( X \) and \( Y \) are marginally independent.
**Definition 10 (Minimum partition)** Let $\mathcal{M}$ be a PDM over a set $N$ of generally dependent variables. A partition $\{N_1, \ldots, N_k\}$ ($k \geq 1$) of $N$ is a minimum partition if the following conditions hold:

1. Any two subsets $N_i$ and $N_j$ ($1 \leq i, j \leq k, i \neq j$) are marginally independent subsets.

2. No $N_i$ ($1 \leq i \leq k$) can be partitioned into marginally independent subsets.

For instance, in the PDM shown in Table 2.2, there exists a minimum partition ($\{a\}, \{c,d\}, \{b\}$).

**Definition 11 [Uncovered colored link]** Let $G$ be a colored I-map of a PI model. Let $l$ be a colored link in a PI submodel $m$ which contains $k_m$ colored links. The link $l$ is uncovered in $m$ if there exists no PI submodel $s$ such that $l$ is also contained (covered) in $s$ and the number of colored links $k_s$ of $s$ satisfies $k_s < k_m$.

![Illustration of covered colored links.](image)

Figure 3.5: Illustration of covered colored links.

Figure 3.5 illustrates uncovered colored links. Given a PDM $\mathcal{M}$, there are two PI submodels $m$ and $s$ which shared three nodes $\{c,d,e\}$. No other PI submodels in $\mathcal{M}$ share nodes with $m$ and $s$. The PI submodel $m$ contains 9 colored links. There are 6 uncovered colored links: $\{(a,c),(a,d),(a,e),(b,c),(b,d),(b,e)\}$, and 3 covered colored links: $\{(c,d),(c,e),(d,e)\}$. The PI submodel $s$ contains smaller
number of colored links. There are 6 colored links and all of them are uncovered: 
\{ (c, d), (c, e), (d, e), (c, f), (d, f), (e, f) \}.

Figure 3.6: Illustration of non covered colored links.

Figure 3.6 illustrates that a PI submodel \( m' \) does not have colored links covered by another PI submodel \( s' \) with same number of colored links. The PI submodel \( m' \) contains 6 colored links \{ (a, b), (a, c), (a, d), (b, c), (b, d), (c, d) \}, and all of them are uncovered. The PI submodel \( s' \) has the same number of colored links \{ (b, c), (b, d), (c, d), (b, e), (c, e), (d, e) \}, and all of them are uncovered.

For two PI submodels in which one PI submodel has colored links covered by the other, the relationship between them has two possibilities: recursive embedding or node sharing. The relationship between the PI submodels over \( N_1 \) and \( N_4 \) in PDM shown in Table 2.2 is an example of recursive embedding. The relationships between the PI submodels \( m \) and \( s \) in Figure 3.5. and \( m' \) and \( s' \) in Figure 3.6 are examples of node sharing. Some numerical examples are presented in Appendix B.

Next, we prove several properties of PI models, which will lead to the proof of the property of algorithm RML.

**Lemma 12** Let \( \mathcal{M} \) be a PI model. The number of colored links \( k \) in \( \mathcal{M} \) satisfies \( k > 1 \).

**Proof:**

34
Let $G_M$ be a minimal colored 1-map of $M$. Suppose in $G_M$ there is only one colored link $(x, y)$. Hence $x$ and $y$ are in two different marginally independent subsets. There exists at least a third variable $z$ in the same PI submodel. Variables $z$ and $x$ are in the same marginally independent subset. Otherwise they are connected by a colored link. Variables $z$ and $y$ can not be connected by a colored link. Hence, we have found $z$ in the same marginally independent subset as $x$ but it is marginally dependent of $y$: a contradiction. □

Lemma 13 Let $M$ be a PI model. If each embedded PI submodel in $M$ contains no more than $i$ uncovered colored links, then $i > 1$.

Proof:

Let $G_M$ be a minimal colored 1-map of $M$. There exists a partial order over all embedded PI submodels according to their number of colored links. Let $s$ be a PI submodel that has the smallest number of colored links. Therefore $s$ has no link covered by any other PI submodels. All the colored links in $s$ are uncovered. Hence the number $k_s$ of colored links in $s$ satisfies $k_s \leq i$. By Lemma 12, $k_s > 1$, therefore $i > 1$. □

Lemma 14 Let $M$ be a partial PI model over $n$ variables. If $M$ has exactly two colored links, then it can have only two marginally independent subsets and $n = 3$.

Proof:

Since a colored link connects two variables in two marginal independent subsets, PI model $M$ with exactly two colored links can have only two marginal independent subsets, denoted as $N_1$ and $N_2$. If there were three or more marginal independent subsets, say, with a third subset $N_3$, then there were at least three colored links among $N_1$, $N_2$ and $N_3$ with each colored link connecting variables in a pair of subsets. This contradicts the condition that $M$ has exactly two colored links. The first part of Lemma 14 is proven.

Suppose $N_1$ contains $m$ ($1 \leq m \leq n - 1$) variables, then $N_2$ contains $n - m$ ($1 \leq n - m \leq n - 1$) variables. Each variable in $N_1$ is connected to each variable in
by a colored link. The number of colored links \( k \) in \( \mathcal{M} \) satisfies \( k = m(n - m) \).

By the condition \( k = 2 \), i.e., \( m(n - m) = 2 \), we have \( n - m = 2/m \). Since \( n - m \geq 1 \), i.e., \( 2/m \geq 1 \), we have \( m \leq 2 \). Therefore \( m = 1 \), or \( m = 2 \). In both cases, \( n = 3 \).

The second part of Lemma 14 is proven. \( \Box \)

**Proposition 15** Let \( PDM \mathcal{M} \) be a PI model and \( \{M_1, \ldots, M_j\} \) \((j \geq 1)\) be the PI submodels in \( \mathcal{M} \). Let \( G_C \) with \( j \) nodes be a direct graph constructed from \( \mathcal{M} \) such that each node in \( G_C \) is labeled by a PI submodel \( M_i \) \((1 \leq i \leq j)\) in \( \mathcal{M} \), and each parent node (PI submodel) has colored links covered by its children. Then \( G_C \) is acyclic and possibly disconnected. We shall call \( G_C \) the PI submodel coverage DAG of \( \mathcal{M} \).

Proof:

We show that there is no directed cycle in \( G_C \). Suppose there is a direct link \((M_x, M_y)\) in \( G_C \). Then \( M_x \) has colored links covered by \( M_y \). By Definition 11, \( M_x \) has more colored links than \( M_y \). There cannot be direct paths from \( M_y \) to \( M_x \). Otherwise \( M_y \) would have more colored links than \( M_x \): a contradiction. Therefore \( G_C \) is a DAG.

\( G_C \) may be disconnected. For instance, the PI submodels in \( \mathcal{M} \) may be partitioned into two or more groups such that no PI submodels in one group can have colored links covered by PI submodels in another group. In this case, \( G_C \) is disconnected.

Each PI submodel without any covered colored links is a leaf in \( G_C \). \( \Box \)

For example, in the PDM shown in Table 2.2, there are four PI submodels over \( N_1, N_2, N_3 \) and \( N_4 \). We can construct the PI submodel coverage DAG shown in Figure 3.7.

Theorem 16 shows the property of algorithm RML.

**Theorem 16** Let \( PDM \mathcal{M} \) be a PI model over \( N \) and \( G_M \) be a minimal colored I-map of \( \mathcal{M} \). Let \( \{N_1, \ldots, N_k\} \) \((k \geq 1)\) be the minimum partition of \( N \). If variables in each subset \( N_j \) \((1 \leq j \leq k)\) satisfy composition and strong transitivity, and each embedded PI submodel in \( \mathcal{M} \) contains no more than \( i \) \((i \geq 2)\) uncovered colored links, then the algorithm RML with parameter \( i \) will return an I-map of \( \mathcal{M} \).
Proof:

By Theorem 8 [40], single link lookahead search can not add any link between any subsets \( N_j \) and \( N_l \) \((j \neq l)\). Since variables in each subset \( N_i \) satisfy composition and strong transitivity, all the black links covered by this subset can be learned by the initial single link search \( \text{lookahead}(1) \) in the first stage by Corollary 18 [40].

We show that RML will learn every embedded PI submodel in \( \mathcal{M} \). Suppose there are totally \( p \) PI submodels in \( \mathcal{M} \). Let \( G_C \) with \( p \) nodes be the PI submodel coverage DAG of \( \mathcal{M} \). We first show that every leaf in \( G_C \) can be learned. We then show that every parent in \( G_C \) can be learned. Based on these and the fact that there are finit number of nodes in \( G_C \), the theorem will be proven.

Since all colored links in leaves are uncovered, a leaf \( M_q \) \((1 \leq q \leq p)\) in \( G_C \) with \( r \) \((1 < r \leq i)\) colored links can be learned in the first \( \text{lookahead}(r) \) search, if it has no colored links shared by another leaf. If \( M_q \) has some nodes shared by another leaf \( M_{q'} \), then \( M_q \) and \( M_{q'} \) must have the same number of colored links (see Figure 3.6 for example). Otherwise one of them should have colored links covered by the other, which becomes a parent. If the two leaves \( M_q \) and \( M_{q'} \) have the same decreased entropy, the leaf tested first will be learned first. Otherwise the leaf with larger decreased entropy will be learned first. The other will be learned during backtracking.

For a parent \( M_u \), suppose all its children are learned in the middle of a search \( \text{lookahead}(v) \). Suppose \( M_u \) has not been learned yet at this moment and \( M_u \) has
uncovered links. If \( w = v \), it can be learned in the same \( \text{lookahead}(v) \). If \( w < v \), it will be learned by a \( \text{lookahead}(w) \) during backtrackings that follow the \( \text{lookahead}(v) \). If \( w > v \), it will be learned by the first \( \text{lookahead}(w) \) after the \( \text{lookahead}(v) \).

\[
\square
\]

### 3.5 Complexity Analysis

For each pass in an \( i \)-link-only search, \( O(N^{2i}) \) sets of \( i \) links need to be tested, one set at each step. Therefore each pass contains \( O(N^{2i}) \) steps. Since each pass adds one set of \( i \) links, an \( i \)-link-only search contains \( O(\frac{N^2}{i}) \) passes.

Table 3.1 shows the relation among the index \( i \), the number of steps per pass and the number of passes in an \( i \)-link-only search.

Table 3.1: The relation among \( i \), steps per pass and passes in an \( i \)-link-only search.

<table>
<thead>
<tr>
<th>( i )</th>
<th># of steps/pass</th>
<th># of passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( O(N^2) )</td>
<td>( O(N^2) )</td>
</tr>
<tr>
<td>2</td>
<td>( O(N^{2*2}) )</td>
<td>( O(\frac{N^2}{2}) )</td>
</tr>
<tr>
<td>3</td>
<td>( O(N^{2*3}) )</td>
<td>( O(\frac{N^2}{3}) )</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>( k - 1 )</td>
<td>( O(N^{2*(k-1)}) )</td>
<td>( O(\frac{N^2}{k-1}) )</td>
</tr>
<tr>
<td>( k )</td>
<td>( O(N^{2*k}) )</td>
<td>( O(\frac{N^2}{k}) )</td>
</tr>
</tbody>
</table>

In order to derive the upper bound of the total number of passes in a \( k \)-link search, we construct a directed graph such that each node in the graph corresponds to one pass during the search and each arrow indicates the chronological order of successive passes. We shall label each node by the number of links to \( \text{lookahead} \) in the pass. For example, a pass in a single link search will be labeled by 1, and a pass in a double-link-only search will be labeled by 2, etc. A graph so constructed will be a directed chain. For the purpose of a later conversion, nodes with the same label will be drawn at the same level and levels are arranged in the decreasing order of the labels. Figure 3.8 shows such a graph for the execution of a 3-link-search. The four
the number of learning passes. The number of learning passes is bounded by $O\left(\frac{N^2}{i-1}\right)$ according to Table 3.1. Hence each node at level $i$ ($1 < i \leq k$) has $O\left(\frac{N^2}{i-1}\right)$ children.

Next, we derive the number of passes at each level. The number of passes at the level $k$ (top level) is $O\left(\frac{N^2}{k}\right)$. The number of passes at the level $k - 1$ is

$$O\left(\frac{N^2}{k} \cdot \frac{N^2}{k-1}\right) = O\left(\frac{N^{2k+2}}{k(k-1)}\right).$$

The number of passes at the level 2 is

$$O\left(\frac{N^2}{k} \cdot \frac{N^2}{k-1} \cdot \cdots \cdot \frac{N^2}{2}\right) = O\left(\frac{N^{2(k-1)}}{k(k-1) \cdot \cdots \cdot 2}\right).$$

Finally, the number of passes at the level 1 is

$$O\left(\frac{N^2}{k} \cdot \frac{N^2}{k-1} \cdot \cdots \cdot \frac{N^2}{2} \cdot N^2\right) = O\left(\frac{N^{2k}}{k(k-1) \cdot \cdots \cdot 2 \cdot 1}\right).$$

Therefore, according to Table 3.1, the total number of search steps is

$$O(N^{2k}) \cdot O\left(\frac{N^2}{k}\right) + O(N^{2(k-1)}) \cdot O\left(\frac{N^{2k+2}}{k(k-1)}\right) + \cdots + O(N^{2k}) \cdot O\left(\frac{N^{2(k-1)}}{k(k-1) \cdots \cdots 2 \cdots 1}\right) + O(N^2) \cdot O\left(\frac{N^{2k}}{k(k-1) \cdots \cdots 2 \cdots 1}\right) = O\left(\frac{1}{k} + \frac{1}{k(k-1)} + \cdots + \frac{1}{k(k-1) \cdots \cdots 2 \cdots 1}\right).$$
Since the factor \( \left( \frac{1}{k} + \frac{1}{k(k-1)} + \ldots + \frac{1}{k(k-1)\ldots 2\cdot 1} \right) \) is upper-bounded by 1, the total number of search steps in a \( k \)-link-search is \( O(N^{2(k+1)}) \).

In order to complete the complexity analysis, we need to take into account of the complexity of each search step, which is dependent on the choice of scoring metric used in \( \text{lookahead}(i) \). Our implementation is based on algorithm 2.6. The complexity of one search step is

\[
O(n + \eta(\eta \log \eta + 2^n)),
\]

where \( n \) is the number of cases in the dataset and \( \eta \) is the maximum size of cliques. Hence the overall complexity of the algorithm is

\[
O(N^{2(k+1)}(n + \eta(\eta \log \eta + 2^n))).
\]

Compared with the complexity of a straightforward multi-link search algorithm [43]

\[
O(k \cdot N^{2k}(n + \eta(\eta \log \eta + 2^n))),
\]

the complexity of a \( k \)-link-search using RML is higher but not much higher. The benefit of the slightly increased complexity is the capability of learning recursively embedded PI models.
Chapter 4

Implementations

This chapter deals with the implementations of learning algorithm. In section 4.1, the drawbacks of early implementation developed in DOS are discussed. Section 4.2 analyzes why Java is a good choice for system upgrading. In section 4.3, the Java implementation is presented. Subsection 4.3.1 illustrates the overall system structure. Subsection 4.3.2 presents the object-oriented model of belief networks. Subsection 4.3.3 shows the components of graphical user interface with several run-time pictures.

4.1 DOS Implementation

The initial version of learning algorithm was developed in DOS system. It is based on a large function library called WEBWEAVR-II, developed by Dr. Y. Xiang, Department of Computer Science, University of Regina. This function library is accumulated over several years of research and development on many aspects of probabilistic reasoning. It contains about 350 functions, which are grouped into several files. It provides effective support to both faculty and graduate students for rapid prototyping of new research ideas. However, as the library grows larger and larger, the drawback on maintainability and team development with the C-based library becomes more and more apparent. The logic relationships among some functions are not immediately clear to new team members. A slight modification of one function may affect a bunch of other functions. Ignorance of one of these functions may cause
errors that are difficult to debug.

Another drawback of the DOS learning program is that its GUI is platform dependent. Although most of the computing functions can be ported to UNIX with reasonable effect, the Turbo C-based GUI must be rewritten in order to run in UNIX. Due to the limitation of memory management and inadequate graphical interface, DOS system is no longer a good choice for new development. As porting the library to any new language is necessarily an major task, we try to find a suitable programming language to upgrade the system. We found Java to be adequate.

4.2 Why Java

Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, multithreaded, and dynamic language[15].

Implementing learning algorithm by Java has the following advantages:

1. Maintainable.

Java provides all the benefits of an object-oriented language. Code written in Java can be reused through inheritance without sacrificing the functionality of already-implemented classes. Modification within one class does not need to modify other classes as long as the external feature of the class is maintained. Using the object-oriented approach, the data in belief network can be designed as a group of classes. The inheritance relations among these classes form a tree structure which will be described in the next section.

2. Portable (Platform independent).

Java is architecture neutral. A same Java application can run on different platforms without modifying the source code. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly. Therefore, when porting the learning program to different platform, the source codes in GUI do not need to be rewritten.
Up to now, the only disadvantage of Java language is low running speed. The source codes of a Java program will be compiled to the intermediate bytecodes, instead of machine codes. When this Java program starts running, the Java bytecodes have to be interpreted to machine codes during the running process. Therefore, running a Java program is slower than running programs compiled directly to machine codes. Since our program is used as a research tool, the performance issue is not crucial. We found that the Java learning program is only slightly slower than its DOS version.

4.3 Java Implementation

Our Java learning program is implemented in Java JDK1.1.3. This program can learn DMNs from datasets and dynamically display the learning process, display a Markov network, and modify the network structure by moving nodes. In the following subsections, we present the system structure of Java learning program, the object-oriented model of belief networks, and the graphical user interface.

4.3.1 System structure

Figure 4.1 shows the system structure of Java learning program. It consists of two parts: Graphical user interface (GUI) and internal network. The GUI part deals with the interaction between user and learning program. The internal network part deals with the computation on belief networks. These two parts communicate through an object called NetworkManager. It manages the communication between objects in GUI part and objects in internal network.

When the program runs, an object called Client will be created. This object creates the objects in GUI part. One of these objects (called NetworkPanel) then create the NetworkManager. The NetworkManager will create an object MarkovNetwork. The MarkovNetwork then create necessary objects in internal network.
4.3.2 Object-oriented model

Alan Kay[12] summarized five basic characteristics of Smalltalk, the first successful object-oriented language and one of the languages upon which Java is based. This represents a pure approach to object-oriented programming:

1. **Everything is an object.** Think of an object as a fancy variable: it stores data, but you can also ask it to perform operations on itself by making requests. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

2. **A program is a bunch of objects telling each other what to do by sending messages.** To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a function for a particular object.

3. **Each object has its own memory made up of other objects.** Or, you make a new type of object by making a package containing existing objects.
Thus, you can build up complexity in a program while hiding it behind the simplicity of objects.

4. **Every object has a type.** Using the parlance, each object is an *instance* of a *class*, where "class" is synonymous with "type". The most important distinguishing characteristic of a class is "what message can you send to it?"

5. **All objects with the same super class can receive the same messages.**

   For example, because an object of type *DNode* (node in a direct graph) is also an object of type *Node*, a DNode is guaranteed to receive Node messages. This *substitutability* is one of the most powerful concepts in OOP.

   A belief network (Bayesian network, Markov network and its Join forest) \((G, P)\) is a graphical representation of the probabilistic dependence relationships among a set of variables in a problem domain, where \(G\) is a graph that represents the network structure and \(P\) is the probability distribution of the network. For instance, a Bayesian network over \(N\) variables is a pair \((G, P(N))\), where \(G\) is a DAG and \(P(N)\) is the probability distribution of BN. A DMN is a pair \((G, P)\) where \(G\) is a chordal graph and \(P\) is the distribution of DMN. A Join forest consists of a hypergraph (see definition in section 2.1) of DMN and a probability table for each clique node in the hypergraph. Therefore, each type of graph can be abstracted to a super (base) classes, and each corresponding belief network can be extended (derived) from it as subclasses.

   Figure 4.2 shows the object-oriented model of internal network. A solid arrow means that the class at the head of the arrow is extended from the class at the tail of the arrow. A dashed arrow means that the class at the head of the arrow is an array of the class at the tail of the arrow.

   A graph consists of nodes and links. A link can be represented by a pair of nodes that form this link. Hence nodes are the basic elements in a graph. In the internal network, there are three types of graphs: *direct graph*, *undirected graph* and *hypergraph*. The common features of a node in these graphs are its label and position. Therefore we define a class *Node* as the base class of nodes in any type of graph. Class *Node*
contains two variables *label* and *position*, and a group of methods to deal with the operations on its variables.

For nodes in each type of graph, we define a subclass that is extended from the same base class *Node*.

1. **DNode**: the class definition of nodes in direct graph. Each node stores its *parent* and *children* nodes in two array *pls* and *cls*.

2. **UNode**: the class definition of nodes in undirect graph. Each node stores its *neighbor* nodes in an array *nls*.

3. **HNode**: the class definition of nodes in hypergraph. Each node in hypergraph is a clique in the corresponding undirect graph. Its local variables store the information about clique members, neighbors, etc.
Based on the above classes, a direct graph is defined as class \textit{DirectGraph}, which is an array of \textit{DNodes}. The class \textit{UndirectGraph} is an array of \textit{UNodes}. The class \textit{HyperGraph} is an array of \textit{HNodes}. The graphical operations such as changing the position of a node, converting graph from one type to another are implemented as the methods in these classes.

Up to now, only the classes for constructing graph structures have been defined. For a belief network over a PDM, variables in the PDM corresponds to nodes in the graph structure. The belief information of a variable should be stored in its corresponding node in the graph. Therefore, we define three types of subclasses of nodes in belief networks. These subclasses are extended from the above three nodes classes respectively:

1. \textit{BNode}: a subclass of \textit{DNode}. Each node stores its corresponding variable states and its belief table.

2. \textit{MNode}: a subclass of \textit{UNode}. Each node stores its corresponding variable states.

3. \textit{CNode}: a subclass of \textit{HNode}. Each node stores two belief tables. One table is for its corresponding clique, the other one is for a sepset that is selected to store in this node.

Similarly, we define the classes of belief networks that are extended from their corresponding graphs. A Bayesian network is defined as class \textit{BayesianNet}, which is a subclass of \textit{DirectGraph}. It is also an array of \textit{BNodes}. The class \textit{MarkovNetwork} is an array of \textit{MNodes} and extended from \textit{UndirectGraph}. The class \textit{JoinForest} is an array of \textit{CNodes} and extended from \textit{HyperGraph}.

The class \textit{CONST} contains several constant variables used in internal network. The class \textit{MATH} contains methods for set operations, float operations and belief table operations. These methods are declared as static such that they can be referred by other classes directly.
Figure 4.3: A run-time image of the graphical user interface

file name can be input or selected in this dialogue box. These two dialogue boxes are designed by Java JDK as standard tools. A third type of dialogue box is created at the beginning of learning process. It's used to edit environment variables. These variables record several important parameters such as the current number of lookahead links $l_k$, the maximum number of lookahead links $maxl_k$, the threshold $alpha$, etc. Figure 4.5 show a run-time image of this dialogue box.
Figure 4.4: Illustration of objects in the graphical user interface.

Figure 4.5: A run-time image of the dialogue box for editing environment variables.
Chapter 5

Experiments and Results

This chapter presents the experiments and the preliminary results. Section 5.1 shows the experimental result of learning ALARM network. Section 5.2 summerizes the results of learning the PDM shown in Table 2.2. In section 5.3, a constructed example of PI model is described. Section 5.4 presents the experimental result of learning from a real dataset. The detailed learning processes are illustrated with several figures.

5.1 The ALARM Network

The ALARM network is constructed by Beinlich [1] as an initial research prototype to model potential anesthesia problems in the operating room. ALARM network contains 37 nodes, and each node has from two to four possible variables. Knowledge for constructing ALARM came from Beinlich’s reading of the literature and from his own experience as an anesthesiologist. It took Beinlich approximately 10 hours to construct the ALARM network structure, and about 20 hours to fill in all the corresponding probability tables.

In order to show the capability of learning PDMs, a dataset of 10000 cases was generated from ALARM. Each case corresponds to a value assignment for each of the 37 variables. Table 5.1 illustrates this dataset.

Before learning, the original datasets are processed by a pre-process program. This program is originally developed by T. Chu and Dr. Y. Xiang. It converts a original
Table 5.1: A dataset generated from ALARM network

<table>
<thead>
<tr>
<th>Case #</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>x37</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>False</td>
<td>Normal</td>
<td>Normal</td>
<td>...</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>Normal</td>
<td>Normal</td>
<td>...</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>Low</td>
<td>Low</td>
<td>...</td>
<td>False</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>False</td>
<td>High</td>
<td>Normal</td>
<td>...</td>
<td>False</td>
</tr>
</tbody>
</table>

dataset into a compressed frequency table. In the experimental study of this thesis research, we further improved the program such that it can select cases that contain no missing values. It can also extract a subset of variables such that the learning program can learn from a submodel in the problem domain.

The compressed frequency table of ALARM is illustrated in Table 5.2. Each variable value in Figure 5.1 is represented by a single character in Table 5.2. For example, the character a for variable x1 stands for False and b stands for True. A frequency column is added in Table 5.2 to indicate the repeated cases. The frequency table is stored in a file whose size is very small compared with the original dataset. For example, the size of original ALARM dataset is 4450K bytes, but the size of its frequency table is only 237K bytes.

Table 5.2: The compressed frequency table of the ALARM network

<table>
<thead>
<tr>
<th>Line #</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>x37</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>...</td>
<td>a</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>...</td>
<td>a</td>
<td>445</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>...</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>5046</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>...</td>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>

One of the successful run used \( k = 1, \delta h = 0.003 \). The learning result is shown in Figure 5.1. The DOS learning program takes 4 hours to finish the learning process.
It's Java version takes double time.

Figure 5.1: The learned Markov network: ALARM

5.2 The PDM in Table 2.2

Our primary emphasis is the capability of learning correctly PDMs with recursively embedded PI submodels. A dataset of 1000 cases was generated from the PDM shown in Table 2.2. The successful run used \( k = 2, \delta h = 0.001 \). The learning process is the same as Figure 3.4. It is summarized in Table 5.3.
Table 5.3: Summary of learning the PDM in Table 2.2

<table>
<thead>
<tr>
<th>i - link-only search</th>
<th>learned link set</th>
<th># graphs tested</th>
<th>cross entropy decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(d, c)}</td>
<td>6</td>
<td>0.0033</td>
</tr>
<tr>
<td>2</td>
<td>{(d, a), (a, c)}</td>
<td>26</td>
<td>0.0139</td>
</tr>
<tr>
<td>2</td>
<td>{(d, b), (b, c)}</td>
<td>29</td>
<td>0.0022</td>
</tr>
<tr>
<td>1</td>
<td>{(a, b)}</td>
<td>30</td>
<td>0.0389</td>
</tr>
</tbody>
</table>

5.3 A Constructed Example: The Music-box Model

In this section, we show the result of learning from a constructed PDM [39], which can be described below:

Three balls are drawn each from a different urn. Urn 1 has 20% white balls and the rest of the balls black. Urn 2 and urn 3 have 60% and 50% of white balls, respectively. A music box plays if all three balls are white or exactly one is white. A dog barks if two random lights are both on or both off. John complains if it's too quiet (neither the box plays nor the dog barks) or too noisy (both the box plays and the dog barks).

The model is specified as a Bayesian network shown in Figure 5.2. Its colored I-map is shown in Figure 5.3.

Figure 5.2: The specification of the music-box model.

The PDM contains five embedded PI submodels over

\[ N_1 = \{\text{ball}_1, \text{ball}_3, \text{music}_\text{box}\}, \quad N_2 = \{\text{ball}_2, \text{ball}_3, \text{music}_\text{box}\}, \]
link was learned: \( L_1 = \{(light1, dog)\} \). It took 28 steps (28 candidate graphs tested). In the second pass, after 27 steps, another link was added: \( L_2 = \{(ball3, music.box)\} \). Note that a standard single-link search learning algorithm will halt and returns this graph which is not an I-map of the domain.

Since nothing was learned in the third pass, a 2-link-only search was performed next. After 884 steps, three sets of links were learned in the following order:

\[
L_3 = \{(light1, light2), (light2, dog)\},
\]

\[
L_4 = \{(ball2, ball3), (ball2, music.box)\},
\]

\[
L_5 = \{(ball1, ball3), (ball1, music.box)\}.
\]

Then the algorithm backtracked to perform a single link search with one link learned: \( L_6 = \{(ball1, ball2)\} \). During the next single link search and the following 2-link-only search, no link was added. Hence a 3-link-only search was performed, which learned the links:

\[
L_7 = \{(music.box, dog), (dog, John), (John, music.box)\}.
\]

The backtracking occurred afterwards, but no more links was learned. Finally, the algorithm halted and returned the correct I-map. A total of 3583 candidate graphs were tested. A summary of the experiment is shown in Table 5.4.

Table 5.4: Summary of learning the music-box model

<table>
<thead>
<tr>
<th>i (-) link (-) only search</th>
<th>learned link set</th>
<th># graphs tested</th>
<th>cross entropy decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( L_1 )</td>
<td>28</td>
<td>0.0822</td>
<td></td>
</tr>
<tr>
<td>1 ( L_2 )</td>
<td>55</td>
<td>0.0069</td>
<td></td>
</tr>
<tr>
<td>2 ( L_3 )</td>
<td>432</td>
<td>0.6109</td>
<td></td>
</tr>
<tr>
<td>2 ( L_4 )</td>
<td>708</td>
<td>0.1922</td>
<td></td>
</tr>
<tr>
<td>2 ( L_5 )</td>
<td>939</td>
<td>0.0146</td>
<td></td>
</tr>
<tr>
<td>1 ( L_6 )</td>
<td>1149</td>
<td>0.4802</td>
<td></td>
</tr>
<tr>
<td>3 ( L_7 )</td>
<td>2327</td>
<td>0.6895</td>
<td></td>
</tr>
</tbody>
</table>
5.4 A Real Dataset: The 1993 GSS on Personal Risk

A real dataset of the 1993 General Social Survey (GSS) on Personal Risk is provided by Dr. Paul Gingrich, Department of Sociology and Social Studies, University of Regina. These data come from Statistics Canada. The survey is a cross Canada survey (excluding residents of the Territories and full-time residents of institutions) conducted from February through December, 1993. The core content for the 1993 GSS, personal risk, includes following subjects: perception of crime, perception of police, perception of courts, crime prevention precautions, frequency of evening activities, accident screening section, criminal victimization screening section, accident report, crime incident report, alcohol and drug use. The detailed information about this survey can be found in [5].

The dataset contains 11960 cases over totally 469 variables. Each variable has two or more states (values). Many variables may have invalid states (missing values). The pre-process program will not count them into the compressed frequency table.

We choose variables on some subjects to explore their probabilistic dependences. Here we present one of the experiments to show the learning capability. Another experiment is presented in Appendix A where a recursively embedded PI submodel is discovered. In subject alcohol and drug use, there is a topic called "Harmful effects of personal drinking", denote as Harmful drinking model. This model contains 8 variables (questions) described in Table 5.5.

Table 5.5: Variables in Harmful drinking model

<table>
<thead>
<tr>
<th>index</th>
<th>Variable</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D15A</td>
<td>Did alcohol harm friendships/social life?</td>
</tr>
<tr>
<td>1</td>
<td>D15B</td>
<td>Did alcohol harm your physical health?</td>
</tr>
<tr>
<td>2</td>
<td>D15C</td>
<td>Did alcohol harm your outlook on life?</td>
</tr>
<tr>
<td>3</td>
<td>D15D</td>
<td>Did alcohol harm your life or marriage?</td>
</tr>
<tr>
<td>4</td>
<td>D15E</td>
<td>Did alcohol harm your work, studies, etc?</td>
</tr>
<tr>
<td>5</td>
<td>D15F</td>
<td>Did alcohol harm your financial position?</td>
</tr>
<tr>
<td>6</td>
<td>D16A</td>
<td>How many drinks should a designated driver have?</td>
</tr>
<tr>
<td>7</td>
<td>D16B</td>
<td>How many drinks should non-designated driver have?</td>
</tr>
</tbody>
</table>
Therefore, this 2-link-only search discovers an embedded partial PI submodel over \( \{1, 5, 7\} \). At this time, the program had tested 426 candidate graphs. Afterwards, the program backtracked to perform a single link search and a double link search. No links were learned by these two searches. The program halted with a connected graph. Totally 788 candidate graphs had been tested. The learning process is shown in Figure 5.6.

Figure 5.6: The process of learning Harmful drinking model.
Chapter 6

Conclusion

6.1 Contribution

PI models escape the detection of many algorithms for learning belief networks that rely on a single link search to detect local dependency. They form a class of difficult PDMs for automated learning. PI models do exist in practice with parity problems and modulus addition problems as special examples [39]. Earlier work by Xiang et al. [43] proposed a straightforward multi-link search algorithm to learn PI models. But some PI submodels may escape this algorithm. In this thesis, we study the relationships among PI submodels, and propose an improved learning algorithm. The main contributions of this thesis are summarized as follows.

- An improved learning algorithm is proposed for learning belief networks in PI domains.

- The algorithm's correctness is analyzed. The analysis shows that it can correctly learn any PDM that contains no PI submodel whose number of uncovered colored links is beyond a predetermined upper bound.

- The algorithm's complexity is analyzed. The analysis shows that the improved learning capability of the new algorithm only causes slight increase in the complexity compared with the straightforward multi-link search algorithm.
- An object-oriented model of belief network is designed. It is used to implement the learning algorithm in Java language. It can be used not only in this learning program, but also in developing other research tools on belief networks.

- PI models are found in practical data. This provides solid evidence that PI models do exist in practice.

6.2 Future Work

The 1993 GSS on Personal Risk data could be further explored by the learning program. Since this dataset contains 469 variables, many subdomains could be considered to study the probabilistic dependences among the domain variables. For example, it's an interesting experiments to explore the probabilistic dependences among variables that cross two or more subjects.

The object-oriented model of belief networks need to be further completed. Since our focus is learning DMNs from data. in the class of Markov network, we only implement necessary methods for learning. For other applications, such as probabilistic inference, the methods should be further implemented.

The interpretation of the PI models found in real data needs to be further studied.
Bibliography


Appendix A

Another Experiment

Another experiment of learning from subdomains in 1993 GSS on Personal Risk is presented here.

We apply the learning algorithm to a topic called Accident Prevention Precautions. Eight variables (questions) are selected from this topic. We shall call the model over these eight variables as Accident Prevention model. The pre-process program get 4303 valid cases from 11960 raw cases. The meaning of each variable is described in Table A.1.

Table A.1: Variables in Accident Prevention model.

<table>
<thead>
<tr>
<th>index</th>
<th>Variable</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A24A</td>
<td>Accident Protection: Use seat belt in vehicle?</td>
</tr>
<tr>
<td>1</td>
<td>A24B</td>
<td>Accident Protection: Wear Helmet riding bicycle?</td>
</tr>
<tr>
<td>2</td>
<td>A24C</td>
<td>Accident Protection: Store medicines from children</td>
</tr>
<tr>
<td>3</td>
<td>A24D</td>
<td>Accident Protection: Use safety equipment?</td>
</tr>
<tr>
<td>4</td>
<td>A25A</td>
<td>Do you have a working smoke detector in your home?</td>
</tr>
<tr>
<td>5</td>
<td>A25B</td>
<td>Do you have a working fire extinguisher at home?</td>
</tr>
<tr>
<td>6</td>
<td>A25C</td>
<td>Do you have first aid supplies at home?</td>
</tr>
<tr>
<td>7</td>
<td>A26</td>
<td>You or household members trained in first aid?</td>
</tr>
</tbody>
</table>

Variables A24A to A24D have the same format of states. They are shown in Table A.2. Note that variables denoted by $M$ are missing values (invalid). Variables A25A, A25B, A25C, and A26 have the same format of states shown in Table A.3.
Table A.2: States format for A24x

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>2</td>
<td>NO</td>
</tr>
<tr>
<td>3</td>
<td>M NOT APPLICABLE</td>
</tr>
<tr>
<td>8</td>
<td>M DO NOT KNOW</td>
</tr>
<tr>
<td>9</td>
<td>M NOT STATED</td>
</tr>
</tbody>
</table>

Table A.3: States format for A25x

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>2</td>
<td>NO</td>
</tr>
<tr>
<td>8</td>
<td>M DO NOT KNOW</td>
</tr>
<tr>
<td>9</td>
<td>M NOT STATED</td>
</tr>
</tbody>
</table>

Successful run using $k = 2$ and $\delta h = 0.0075$. The learning process is shown in Figure A.1. The first single-link lookahead search learns a disconnected graph shown in Figure A.1 G1. In the followed double-link-only search, the following three PI submodels are learned and shown in graphs G2, G3, and G4 respectively.

$$M_1 : \{A24D, A25B, A26\}$$

$$M_2 : \{A24B, A24C, A24D\}$$


Note that the PI submodel $M_1$ is recursively embedded in the PI submodel $M_3$. After this double-link-only search, a backtracking occurs. No links are learned in backtracking and the program halts with a connected graph shown in Figure A.2.
Figure A.1: The process of learning *Accident Prevention* model

Figure A.2: The learned Markov network: *Accident Prevention* model
Appendix B

PI Submodels that Share Nodes

Here we present three examples where two PI submodels share some common nodes. Each example is a PDM over four binary variables \(\{a, b, c, d\}\). In each PDM, an embedded PI submodel \(M_1\) over \(\{a, b, c\}\) and another embedded PI submodel \(M_2\) over \(\{b, c, d\}\) share two nodes \(\{b, c\}\). Nodes \(a\) and \(d\) display different relationships in three examples. Note that in the corresponding PI submodel coverage DAG, nodes \(M_1\) and \(M_2\) are two leaves. Therefore it is possible that two leaves PI submodels share nodes.

Example 17 Table B.1 shows a PDM \(M_\alpha\) over four binary variables \(\{a, b, c, d\}\). A PI submodel \(M_1\) over \(\{a, b, c\}\) and a PI submodel \(M_2\) over \(\{b, c, d\}\) share two nodes \(\{b, c\}\). Variables \(a\) and \(d\) satisfy \(\neg I(a, \phi, d)\).

Example 18 Table B.2 shows a PDM \(M_\beta\) over four binary variables \(\{a, b, c, d\}\). A PI submodel \(M_1\) over \(\{a, b, c\}\) and a PI submodel \(M_2\) over \(\{b, c, d\}\) share two nodes \(\{b, c\}\). Variables \(a\) and \(d\) satisfy \(I(a, \phi, d)\).

Example 19 Table B.3 shows a PDM \(M_\gamma\) over four binary variables \(\{a, b, c, d\}\). A PI submodel \(M_1\) over \(\{a, b, c\}\) and a PI submodel \(M_2\) over \(\{b, c, d\}\) share two nodes \(\{b, c\}\). Variables \(a\) and \(d\) satisfy \(I(a, \{b, c\}, d)\).
Table B.1: Example 17: A model $\mathcal{M}_\alpha$ with two PI submodels sharing nodes

<table>
<thead>
<tr>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 0, 0, 0)$</td>
<td>0.05</td>
<td>$(1, 0, 0, 0)$</td>
<td>0.05</td>
</tr>
<tr>
<td>$(0, 0, 0, 1)$</td>
<td>0.01</td>
<td>$(1, 0, 0, 1)$</td>
<td>0.01</td>
</tr>
<tr>
<td>$(0, 0, 1, 0)$</td>
<td>0.04</td>
<td>$(1, 0, 1, 0)$</td>
<td>0.04</td>
</tr>
<tr>
<td>$(0, 0, 1, 1)$</td>
<td>0.05</td>
<td>$(1, 0, 1, 1)$</td>
<td>0.05</td>
</tr>
<tr>
<td>$(0, 1, 0, 0)$</td>
<td>0.05</td>
<td>$(1, 1, 0, 0)$</td>
<td>0.09</td>
</tr>
<tr>
<td>$(0, 1, 0, 1)$</td>
<td>0.01</td>
<td>$(1, 1, 0, 1)$</td>
<td>0.13</td>
</tr>
<tr>
<td>$(0, 1, 1, 0)$</td>
<td>0.04</td>
<td>$(1, 1, 1, 0)$</td>
<td>0.24</td>
</tr>
<tr>
<td>$(0, 1, 1, 1)$</td>
<td>0.05</td>
<td>$(1, 1, 1, 1)$</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table B.2: Example 18: A model $\mathcal{M}_\beta$ with two PI submodels sharing nodes

<table>
<thead>
<tr>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 0, 0, 0)$</td>
<td>0.03</td>
<td>$(1, 0, 0, 0)$</td>
<td>0.07</td>
</tr>
<tr>
<td>$(0, 0, 0, 1)$</td>
<td>0.006</td>
<td>$(1, 0, 0, 1)$</td>
<td>0.14</td>
</tr>
<tr>
<td>$(0, 0, 1, 0)$</td>
<td>0.021</td>
<td>$(1, 0, 1, 0)$</td>
<td>0.056</td>
</tr>
<tr>
<td>$(0, 0, 1, 1)$</td>
<td>0.03</td>
<td>$(1, 0, 1, 1)$</td>
<td>0.07</td>
</tr>
<tr>
<td>$(0, 1, 0, 0)$</td>
<td>0.07</td>
<td>$(1, 1, 0, 0)$</td>
<td>0.07</td>
</tr>
<tr>
<td>$(0, 1, 0, 1)$</td>
<td>0.014</td>
<td>$(1, 1, 0, 1)$</td>
<td>0.126</td>
</tr>
<tr>
<td>$(0, 1, 1, 0)$</td>
<td>0.056</td>
<td>$(1, 1, 1, 0)$</td>
<td>0.224</td>
</tr>
<tr>
<td>$(0, 1, 1, 1)$</td>
<td>0.07</td>
<td>$(1, 1, 1, 1)$</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table B.3: Example 19: A model $\mathcal{M}_\gamma$ with two PI submodels sharing nodes

<table>
<thead>
<tr>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
<th>$(d, a, b, c)$</th>
<th>$P(d, a, b, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 0, 0, 0)$</td>
<td>0.1 * 0.1/0.24</td>
<td>$(1, 0, 0, 0)$</td>
<td>0.14 * 0.1/0.24</td>
</tr>
<tr>
<td>$(0, 0, 0, 1)$</td>
<td>0.02 * 0.02/0.16</td>
<td>$(1, 0, 0, 1)$</td>
<td>0.14 * 0.02/0.16</td>
</tr>
<tr>
<td>$(0, 0, 1, 0)$</td>
<td>0.08 * 0.08/0.36</td>
<td>$(1, 0, 1, 0)$</td>
<td>0.28 * 0.08/0.36</td>
</tr>
<tr>
<td>$(0, 0, 1, 1)$</td>
<td>0.1 * 0.1/0.24</td>
<td>$(1, 0, 1, 1)$</td>
<td>0.14 * 0.1/0.24</td>
</tr>
<tr>
<td>$(0, 1, 0, 0)$</td>
<td>0.1 * 0.14/0.24</td>
<td>$(1, 1, 0, 0)$</td>
<td>0.14 * 0.14/0.24</td>
</tr>
</tbody>
</table>