A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF INFORMATION SCIENCE

IN THE GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY OF

HOKKAIDO UNIVERSITY

平成 30 年度　博士論文

Fast Exact Inference Algorithms for Bayesian Networks Based on ZDD Operations

ZDD 演算に基づくベイジアンネットワークの高速かつ厳密な

確率推論アルゴリズム

Shan Gao

高サン

Large-Scale Knowledge Processing Laboratory

Division of Computer Science and Information Technology

Graduate School of Information Science and Technology

Hokkaido University

北海道大学　大学院情報科学研究科

情報理工学専攻　大規模知識処理研究室

November 2018

Abstract

Compiling Bayesian networks (BNs) into secondary structures to implement efficient exact inference is a hot topic in probabilistic modeling. One class of algorithms to compile BNs is to transform the BNs into Zero-Suppressed Binary Decision Diagrams (ZDDs) to perform efficient exact inference. This method has attracted much attention. A ZDD is a data structure for manipulating boolean functions and item combinations efficiently. By compiling a BN into ZDDs, computation time for exact inference using ZDDs is reduced to linear time in the size of the ZDDs. Also, cache memory techniques further help to accelerate the inference. However, as the size of BN grows, compiling ZDDs becomes unacceptable in both time consumption and ZDD size which hinders BN practical applications. In this thesis, we focus on improving the ZDD-based method to efficiently execute the exact inference of BNs. We take into account two aspects, condensing ZDD size through factorizations and compiling decomposition forms of BNs instead of the whole networks.

In Chapter 3, to condense ZDD size, we propose a fast factorization method based on the d-separation structures in BNs to factor the ZDDs with large size into small ones. The weak divison algorithm is used to factorize a large sum-of-product into several compact sum-of-products. It is known as

the most successful and prevalent technique of logic synthesis and optimization. Minato et al. proposed an improvement of this algorithm, known as the fast weak division algorithm for ZDD-based logic operations. In this algorithm, variables appearing many times are iteratively extracted and then used as *divisors* to factorize a ZDD. We can use this algorithm to factorize a large ZDD into small ones. However, for a ZDD representing a BN, the approach to use every multiply appearing variables as divisors to factorize a ZDD leads to unacceptable time consumption for factorization. We improve this factorization algorithm by extracting divisors using d-separations in BNs. In our method, variables appearing multiple times are extracted once so that time consumption are largely reduced. What is more, the resulting ZDD is largely condensed which would result in big improvements in time of exact inference.

In Chapter 4, we propose a fast message passing algorithm using ZDD-based local structure compilation. The idea of factorizing ZDDs is based on the fact that we are able to generate ZDDs for a BN. As BN gets large, the size of resulting ZDD will be too large to generate at the first place. Therefore, we consider to compile the decomposition form of a BN instead of a whole BN into ZDDs. A junction tree is one of the most prevalent decomposition forms of a BN whose node is a clique consisting of BN vertexes. Performing the message passing algorithm on a junction tree for exact inference is currently one of the most prominent BN inference algorithms. The algorithm works by passing real-valued functions called *messages* along with the edges between the two nodes in a junction tree. The performance of this algorithm depends on a BN's treewidth or the optimal maximal clique size of a corresponding junction tree. In our method, a junction tree is directly compiled into ZDDs. We introduce *message variables* in ZDDs to pass messages. Then, the message passing algorithm can be performed on ZDDs. By utilizing techniques of node sharing

and cache memory in ZDDs, parameters in BNs which share the same value can compactly represented. Moreover, repetitive local computations during the message passing algorithm are avoided. Our method of conducting message passing on ZDDs performs much faster than the performing on the original junction tree which is generated through a well known heuristic way called *min-fill* method.

In Chapter 5, we present the method of separate compilation of BNs for efficient exact inference. We propose to combine these two approaches that using the d-separation structure in BNs to partition a large BN into several components. For every given BN, serial pattern d-separation sets are found and used to partition the BN into conditionally independent components. Then we compile these components into small ZDDs and perform exact inference using these ZDDs. Separately compiling these components into ZDDs is more efficient than generating giant ZDDs for a whole network. However, partitioning a BN into too many components may give rise to considerable time consumption which grows exponentially with the number of vertexes in serial pattern d-separations. To trade off the off-line time consumption (for finding d-separations and compiling ZDDs) and on-line time consumption (for inference using ZDDs), the d-separations used to partitioning BNs are restricted to one-vertex and found using Tarjan's vertex-cut algorithm which can be performed linear time in the number of BN vertexes. The experiments illustrate that one-vertex d-separations exist in most BNs. Partitioning BNs with one-vertex d-separations improves the speed for both compilation and inference significantly than the conventional ZDD-based method.

In Chapter 6, we conclude our remarks and discuss the future work and open problems. We show that ideas in this thesis are valuable since not only for

ZDDs, they are also usable for other data structures. We hope to improve other logic operation based approaches such as *d-DNNF*, an efficient logic circuit used in BN inference recently. We expect that using fast logic operations can bring a big improvement for the exact inference of BNs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Probabilistic graphical models (PGMs), which are widely used as statistical analysis tools that estimate the probability of events occurring on the basis of historical data, have become more and more popular [5, 31, 51, 17]. A PGM is a compact or factorized representation of a set of independencies over a set of random variables. One branch of PGMs, namely, Bayesian networks (BNs) are commonly used in the artificial intelligence [2, 7, 43, 44]. A BN is a compact representation of joint probability distributions of a set of random variables. Their conditional dependencies are explicitly expressed via a directed acyclic graph (DAG). Usually, we have prior knowledge and some observations over variables. Then we want to infer the posterior knowledge of variables based on these observations to find the most possible reason that leads to these observations. Such tasks are called *exact inference* which is NP-hard even in the simple case that the structure of a BN is singly connected [27, 54, 24]. A great amount

of efforts has been tried to improve the efficiency of exact inference.

Inference with a *junction tree* (also known as *jointree*) transformed from a BN, is currently among the most prominent BN inference algorithms [15, 28, 36, 53]. The junction tree transformed from a given BN is a tree where each node and edge in this tree is labeled by a set of variables of the given BN [32]. Inference using junction trees is executed through a well-known message passing procedure. Given some evidence (observed variables), the junction tree algorithm propagates messages (information of evidence) between nodes, also indicated as *clusters*. After a full round of message passing, one can compute the marginal probability for every cluster. The junction tree is not an arbitrary tree of clusters as it must satisfy some conditions. Firstly, each variable and its parents in a BN are treated as a *family*. Every family in the BN must belong to some tree cluster. Secondly, if a variable of the BN appears in two clusters in the tree, it must also appear in every cluster on the path connecting them. These two conditions legitimize the message passing algorithm. Ensuring these conditions may lead to clusters that are large. The performance of junction tree algorithms depends on the BN's treewidth, the optimal maximum cluster size [37]. Thus, how to construct a junction tree with minimum width becomes the main issue [4, 47]. However, this is also an NP-hard problem.

To break the barrier of treewidth in the exact inference, approaches of using structured representations, such as logical formalism, to compactly represent the probability distribution of BNs have been proposed [11]. The main breakthrough is to exploit the local structures in BNs. Local structures [9], which refer to the specific properties attained by the probabilities quantifying the network, can imply independence that is not visible at the structural

level. Such independence may be utilized computationally. Using structured representations allow the inference to be performed quite efficiently. Time consumption for inference of these methods strongly depends on the size of data structure into which a BN is compiled. Approaches of compiling BNs into structured representations such as, Zero-suppressed Binary Decision Diagrams (ZDDs) [39], Affine Algebraic DDs (AADDs) [50] and Probabilistic Sentential Decision Diagrams (PSDDs) [30] have been widely used for efficient exact inference in BNs. Experiment results of these compilation approaches have shown that they yield significant time and space savings over the conventional tabular representations on probability computations [28, 11, 42, 50]. In this thesis, we mainly deal with the ZDD-based compilation method and propose methods to improve the exact inference based on ZDD's logic operations.

A BN can be characterized by an *Multi-Linear Function* (MLF) [12]: a polynomial combination of propositional variable sets corresponding to realizations of random variables. The ZDD-based method compiles the MLF into ZDDs to compactly represent the BN. Each ZDD represents the MLF of a BN vertex. The ZDD for a BN vertex depends only on the ZDDs of its ancestor vertices. Through node-sharing techniques in ZDDs, MLFs can be compactly represented. Exact inference is carried out in a time almost linear in the ZDD size based on the multi-value multiplication algorithm of ZDDs [42]. Node sharing techniques of ZDDs help exploit local structures such as different variables sharing the same value and local computation [33, 18, 8, 19]. Cache memory techniques of ZDDs further accelerate the inference significantly [42]. However, since the size of MLFs is exponential to the number of variables, when the BN gets large, the size of ZDDs becomes unacceptable. Furthermore, considerable time for compiling a BN into ZDDs puts limits to the usefulness of the ZDD-based compilation approach.

1.2 Our Contributions

In this thesis, we focus on improving the conventional ZDD-based method to efficiently execute the exact inference of BNs. We take into account two aspects, condensing ZDD size through factorizations and compiling decomposition forms of BNs instead of the whole networks. To condense he ZDD size, we improve the existing ZDD factorization method to factor the ZDDs with large size into small ones. Through our factorization algorithm, the ZDD size compiling a BN is largely condensed so that exact inference is expected to be accelerated. Secondly, we proposed to directly compile a junction tree into ZDDs. Then, the message passing algorithm is conducted on ZDDs. By utilizing techniques of node sharing cache memory in ZDDs, parameters in BNs which share the same value can be compactly represented. Moreover, repetitive local computations during the message passing algorithm are avoided. Finally, we proposed a method which combines these two approaches. First, a large BN is partitioned into several conditionally independent components. Then, we separately compile these components into ZDDs. For every given BN, a set of vertices are found in linear time to partition the BN into conditionally independent components. Then we compile these components into small ZDDs and perform exact inference using these ZDDs. Separately compiling these components into ZDDs is more efficient than generating a giant ZDD for a whole network. Partitioning BNs into several components improves the speed for both compilation and inference largely than the conventional ZDD-based method.

The idea of separately compiling BNs in this thesis is valuable, since it also fits other logic-based data structures. We expect that using fast logic

operations can bring a big improvement for the exact inference of BNs.

1.3 Related Works

Darwiche [12] proposed the MLFs to represent the joint distribution induced by a BN and compile MLFs into logical formalism to exploit local structures. In their method, they compile a BN into conjunctive normal form (CNF) and then compile the CNF into smooth deterministic negation normal form (d–DNNF). They claimed that time consumption for probability calculation using a d-DNNF is strongly related to the size of the d-DNNF. This approach has been shown successful in a number of instances where other algorithms that utilize local structures are overwhelmed [13, 6].

1.4 Structure of This Thesis

The rest of this thesis is organized as follows. Chapter 2 provides preliminaries of this thesis, such as knowledge about probability theory, Bayesian networks and ZDDs. We also introduce basic definitions and notations we use in this thesis. In Chapter 3, we show our method of efficiently factorizing ZDDs into small ones. In Chapter 4, we explain how to compile a junction tree with ZDDs and perform message passing on ZDDs. The approach of separately compiling BNs is presented in Chapter 5. Control experiments with the efficient *d-DNNF*-based compilation method are exhibited in this chapter. Finally in Chapter 6, we conclude our remarks and discuss the open problems.

Chapter 2

Preliminaries

In this chapter, we introduce the basic background knowledge about probability theory, such as probabilities, random variables and independence among random variables. Based on these basic knowledge, the definition of BNs is given and the main task of exact inference in BNs is explained. We also present how to transform a BN into a multi-linear function and then compile this BN using ZDDs. Inference can be performed with ZDD logic operations in almost linear time in ZDD size. Thus, condensing ZDD size to accelerate inference is the main issue in this thesis. We also introduce notations we use in the later chapters.

2.1 Probability Theory

The word "probability" we use nowadays, is used to describe our belief in some uncertainties, such as "How possible it will rain tomorrow since we felt the strong wind this afternoon". Probability theory [16, 3, 22, 29] provides the rules one should obey and the formal foundations when discussing such estimates [31]. Before we come to the definition of probability, we first declare the basic knowledge about probability. In what follows, we denote by $\mathbb{N} = \{0, 1, \dots\}$ the set of all non-negative integers, and by $\mathbb{R}$ the set of all real numbers.

2.1.1 Events and Probabilities

Suppose an experiment of tossing a coin three times under exactly the same conditions. All the possible outcomes of this experiment would be {HHH,HHT,HTH,HTT,THH,THT,TTH,TTT} (where "H" refers to Head and "T" refers to Tail). We refer the statement of "get two heads in the first and second tossing and one tail in this last tossing" as an event which describes an outcome of the experiment. There can be other statements which corresponds to some outcomes. For example, the event "two heads and one tail" is related to the outcomes of {HHT, HTH, THH}. Formally, we use a countable set Ω, called a sample space, to indicate all the possible outcomes of a random experiments.

Definition 1 An event space is a set $S \subset 2^{\Omega}$ of events satisfying the following conditions (i)–(iii):

(i) $\emptyset \in \boldsymbol{S}$ and $\Omega \in \boldsymbol{S}$ hold.

(ii) For any countably many events $\alpha_0, \ldots, \alpha_i, \cdots \in \boldsymbol{S}$ $(i \in \mathbb{N})$, $(\alpha_0 \cup \cdots \cup \alpha_i \cup \ldots) \in \boldsymbol{S}$ holds.

(iii) For any event $\alpha \in \boldsymbol{S}$, then $\Omega \setminus \alpha \in \boldsymbol{S}$ holds.

In this thesis, we restrict ourselves to finite and discrete sample spaces.

Definition 2 The probability function is a function $P : \boldsymbol{S} \to \mathbb{R}$ that maps each event $\alpha \in \boldsymbol{S}$ to a real number $P(\alpha) \in \mathbb{R}$ that satisfies the following conditions (i)–(iii):

(i) For any event $\alpha \in \boldsymbol{S}$, $0 \leq P(\alpha) \leq 1$ holds.

(ii) $P(\Omega) = 1$ holds.

(iii) If countably many events $\alpha_0, \ldots, \alpha_i, \cdots \in \boldsymbol{S}$ $(i \in \mathbb{N})$ are mutually exclusive, then $P(\alpha_0 \cup \cdots \cup \alpha_i \cup \ldots) = P(\bigcup_{i \in \mathbb{N}} \alpha_i) = \sum_{i \in \mathbb{N}} P(\alpha_i)$ holds.

There are many interpretations for probabilities [31]. In the coin tossing experiment, we treat probabilities as frequencies of events. Intuitively, the probability $P(\alpha)$ of an event α quantifies the degree of confidence that α will occur. If $P(\alpha) = 1$, we are certain that one of the outcomes in α occurs, and if $P(\alpha) = 0$, we consider all of them impossible. Other probability values represent options that lie between these two extremes. For $P(\alpha) = 0.3$, the probability of an event is the fraction of times the event occurs if we repeat the experiment indefinitely.

2.1.2 Rules in Probabilities

There are some important rules in probabilities we should obey when estimating the uncertain events.

Conditional Probabilities: Knowing that α happens, how do we change our belief on β occurring? The answer can be represented with the notion of conditional probability. Formally, the conditional probability of β given α is defined as:

$$P(\beta \mid \alpha) = \frac{P(\alpha \cup \beta)}{P(\alpha)},\ P(\alpha) \neq 0. \tag{2.1}$$

Chain Rule: we can express the probability of a combination of several events in terms of the probability of the first, the probability of the second given the first, and so on.

$$P(\alpha \cap \beta) = P(\beta \mid \alpha)\, P(\alpha)\,. \tag{2.2}$$

More generally, if $\alpha_1, \ldots, \alpha_k$ are events, then we can write:

$$P(\alpha_1 \cap \cdots \cap \alpha_k) = P(\alpha)\, P(\alpha_2 \mid \alpha_1) \ldots P(\alpha_k \mid \alpha_1 \cap \cdots \cap \alpha_{k-1})\,. \tag{2.3}$$

It is important to notice that we can expand this expression using any order of events.

Bayes' Rule: it is a very important rule that allows us to compute the posterior probability P(α | β) observing α happen through conditional probability P(β | α) and $P(\alpha)$ and $P(\beta)$ which come from our prior knowledge for

α and β.

$$P(\alpha \mid \beta) = \frac{P(\beta \mid \alpha)\, P(\alpha)}{P(\beta)},\ P(\beta) \neq 0. \tag{2.4}$$

Usually, given some observations and hypothesis, calculating posterior probabilities is called *probability inference*.

2.1.3 Random Variables and Independence

Random variables can be used to discuss attributes of outcomes. It is a function from the sample space Ω to the real numbers. Random variables can be discrete or continuous. In this thesis, we only discuss discrete random variables. Once we define a random variable X, we can consider the distribution for X. In this thesis, we use X_i to represent a random variable. Every random variable has a discrete domain of $\mathrm{Dom}(\{X_i\}) = \{x_{i,1}, x_{i,2}, \ldots, x_{i,k_i}\}$. There are some distributions used a lot. The *marginal probability* of X: the probability distribution over events that can be described using X. In many situations, we are interested in questions that involve the values of several random variables which indicate the *joint distribution* written as $P(X_1, X_2)$: Usually, we use $P(X_1 = x_{1,1}, X_2 = x_{2,1})$ denotes likelihood of two or more random variables occurring together .

One of the most important conception in probability theory is the conditional independence. Given three events α, β and γ, we say α is conditionally independent of β given γ if $P(\alpha \mid \beta \cap \gamma) = P(\alpha \mid \gamma)$ or if $P(\beta \cap \gamma) = 0$. This means that if γ occurs, whether β occurs or not will not change my belief for α occurring. Similarly, we have the conditional independence for variables as: For three random variables X_1, X_2, and X_3. we say that X_1 is conditionally independent of X_2 given X_3 is instantiated if we have:

$P(X_1 = x_{1,k_1} \mid X_2 = x_{2,k_2}, X_3 = x_{3,k_3}) = P(X_1 = x_{1,k_1} \mid X_3 = x_{3,k_3})$ or $P(X_3 = x_{2,k_2} \cap X_3 = x_{3,k_3}) = 0$ for all $x_{1,k_1} \in \text{Dom}(\{X_1\})$, $x_{2,k_2} \in \text{Dom}(\{X_2\})$, $x_{3,k_3} \in \text{Dom}(\{X_3\})$.

2.2 Bayesian Networks and Exact Inference

2.2.1 Bayesian Networks

In a realistic world, there can be hundreds of factors, in terms of variables. It is difficult to direct calculate posterior probabilities using probability rules with so many variables. There is a straightforward algorithm for computing them which is to using the joint distribution of all related variables. Then summing out the irrelevant variables. The table size for represent the joint distributions grows exponentially in the number of variables assuming that every variable is binary-valued. Such calculation has exponential space and time complexity. BNs were developed to compactly represent joint distributions over a large number of variables using little space by exploiting conditional independencies. We are also able to perform probabilistic inference with BNs in an acceptable amount of time [45].

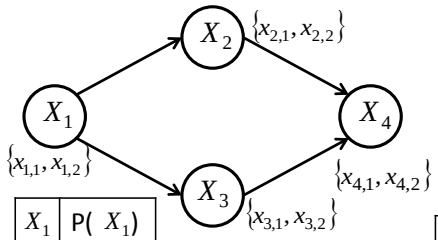
A *Bayesian Network* (BN) [49] is a Directed Acyclic Graph (DAG) which defines a joint distribution over a set of random variables. A BN is defined as $B \triangleq \langle G, \Theta \rangle$, where $G \triangleq \langle V, A \rangle$ is a DAG that each vertex represents a random variable and arcs between vertexes represent the dependencies among corresponding variables. We use $V \triangleq \{X_1, \ldots, X_n\}$ to represent all the n vertexes and $A \subset V \times V$ as the set of directed arcs that represent dependencies between vertexes. The set of parents of X_i in a BN is defined as $\Pi_i \triangleq \{X_{i'} \in V \mid (X_{i'}, X_i) \in A\}$. The second parameter $\Theta \triangleq \{\theta_{i,j,k}\}_{i,j,k}$

denotes a set of conditional probabilities of vertexes, where $\theta_{i,j,k}$ indicates the probability of X_i taking the k-th value $x_{i,k}$ given its set of parents Π_i taking the j-th instantiation $\pi_{i,j}$: $\theta_{i,j,k} \triangleq P(X_i = x_{i,k} \,|\, \Pi_i = \pi_{i,j})$. A BN assumes that each vertex is independent of its non descendent vertices when its parents are instantiated. Thus, given a BN B, the joint distribution of vertexes in V defined by B is represented as:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid \Pi_i). \tag{2.5}$$

A BN which consists of vertexes $\{X_1, X_2, X_3, X_4\}$ is shown in Fig 2.1. The conditional probability distribution (CPD) for every vertex is presented in tables known as *Conditional Probability Tables* (CPTs). Parameters such as $\theta_{4,1,1}$ for X_4 means that X_4 takes its first value $x_{4,1}$ with its parents set $\{X_2, X_3\}$ instantiated with the first instantiation $\{(x_{2,1}, x_{3,1})\}$. Similarly, $\theta_{4,4,2}$ means that X_4 takes the second value $x_{4,2}$ with its parents instantiated with the fourth instantiation $\{(x_{2,2}, x_{3,2})\}$.

In this thesis, a boldface letter $\boldsymbol{X} \subset V$ indicates a set of vertexes. Its domain is referred as $\mathrm{Dom}(\boldsymbol{X}) = \{\boldsymbol{x}_l\}_{l \in \mathbb{N}}$ where $\boldsymbol{x}_l$ is the l-th instantiation of $\boldsymbol{X}$ and l ranges from 1 to the total number of instantiations of variables in $\boldsymbol{X}$. For the BN in Fig 2.1, for each vertex, we have $\mathrm{Dom}(\{X_1\}) = \{x_{1,1}, x_{1,2}\}$, $\mathrm{Dom}(\{X_2\}) = \{x_{2,1}, x_{2,2}\}$, $\mathrm{Dom}(\{X_3\}) = \{x_{3,1}, x_{3,2}\}$, $\mathrm{Dom}(\{X_4\}) = \{x_{4,1}, x_{4,2}\}$. Also, for any given vertex set such as $\{X_2, X_3\}$, $\mathrm{Dom}(\{X_2, X_3\}) = \{(x_{2,1}, x_{3,1}), (x_{2,1}, x_{3,2}), (x_{2,2}, x_{3,1}), (x_{2,2}, x_{3,2})\}$. The joint distribution for vertexes in $\boldsymbol{X}$ is represented as $P(\boldsymbol{X})$. $P(\boldsymbol{X} = \boldsymbol{x}_l)$ indicates the probability of $\boldsymbol{X}$ instantiated with $\boldsymbol{x}_l$. Usually, we write $P(\boldsymbol{x}_l)$ for short. We use the notation $\mathrm{Ances}(\boldsymbol{X})$ to represent all the ancestor vertexes of $\boldsymbol{X}$ in the BN: $\mathrm{Ances}(\boldsymbol{X}) = \{X_i \mid$ at least one of the vertexes in $\boldsymbol{X}$ is reachable



X_1	$P(X_1)$
$x_{1,1}$	$\theta_{1,1,1} = 0.4$
$x_{1,2}$	$\theta_{1,1,2} = 0.6$

$X_2 \mid X_1$	$P(X_2 \mid X_1)$
$x_{2,1} \mid x_{1,1}$	$\theta_{2,1,1} = 0.8$
$x_{2,2} \mid x_{1,1}$	$\theta_{2,1,2} = 0.2$
$x_{2,1} \mid x_{1,2}$	$\theta_{2,2,1} = 1$
$x_{2,2} \mid x_{1,2}$	$\theta_{2,2,2} = 0$

$X_3 \mid X_1$	$P(X_3 \mid X_1)$
$x_{3,1} \mid x_{1,1}$	$\theta_{3,1,1} = 0.5$
$x_{3,2} \mid x_{1,1}$	$\theta_{3,1,2} = 0.5$
$x_{3,1} \mid x_{1,2}$	$\theta_{3,2,1} = 0.5$
$x_{3,2} \mid x_{1,2}$	$\theta_{3,2,2} = 0.5$

$X_4 \mid X_2, X_3$	$P(X_4 \mid X_2, X_3)$
$x_{4,1} \mid x_{2,1}, x_{3,1}$	$\theta_{4,1,1} = 0.0$
$x_{4,2} \mid x_{2,1}, x_{3,1}$	$\theta_{4,1,2} = 1$
$x_{4,1} \mid x_{2,2}, x_{3,1}$	$\theta_{4,2,1} = 0.2$
$x_{4,2} \mid x_{2,2}, x_{3,1}$	$\theta_{4,2,1} = 0.8$
$x_{4,1} \mid x_{2,1}, x_{3,2}$	$\theta_{4,3,1} = 0.8$
$x_{4,2} \mid x_{2,1}, x_{3,2}$	$\theta_{4,3,2} = 0.2$
$x_{4,1} \mid x_{2,2}, x_{3,2}$	$\theta_{4,4,1} = 0.4$
$x_{4,2} \mid x_{2,2}, x_{3,2}$	$\theta_{4,4,2} = 0.6$

Figure 2.1 An example of BN

from $X_i\}$. Thus, we have $\text{Ances}(\{X_4\}) = \{X_1, X_2, X_3\}$.

2.2.2 Exact Inference in Bayesian Networks

Given a BN representing a joint distribution over all the vertexes, the probability of any given vertex set $\boldsymbol{X} \subset V$ instantiated with $\boldsymbol{x}_l$ can be calculated by summing over all possible instantiations of all other vertexes:

$$P(\boldsymbol{x}_l) = \sum_{V \setminus \boldsymbol{X}} P(X_1, \ldots, \boldsymbol{X} = x_l, \ldots, X_n)\,, \quad \text{where } V \setminus \boldsymbol{X} \text{ means the complement of } \boldsymbol{X} \text{ in } V. \tag{2.6}$$

Usually, $\sum_{X_i}$ refers to summing over all possible values that X_i can take. We use notation $\sum_{\boldsymbol{X}}$ as a shorthand for $\sum_{X_1} \sum_{X_2} ... \sum_{X_i}$, summing over all vertexes in $\boldsymbol{X} = \{X_1, X_2, ..., X_i\}$ For example, the probability of $P(x_{4,2})$ in Fig 2.1 can be computed as:

$$\sum_{X_1, X_2, X_3} P(X_1)\, P(X_2 \mid X_1)\, P(X_3 \mid X_1)\, P(X_4 = x_{4,2} \mid X_2, X_3). \tag{2.7}$$

Usually, the task of *exact inference* in a BN is to calculate the probability of some instances $\boldsymbol{x}_l^{(1)}$ given some observations $\boldsymbol{x}_k^{(2)}$: $P\left(\boldsymbol{x}_l^{(1)} \mid \boldsymbol{x}_k^{(2)}\right)$. According to equation (2.1), we have:

$$P\left(\boldsymbol{x}_l^{(1)} \mid \boldsymbol{x}_k^{(2)}\right) = \frac{P\left(\boldsymbol{x}_l^{(1)}, \boldsymbol{x}_k^{(2)}\right)}{P\left(\boldsymbol{x}_k^{(2)}\right)}. \tag{2.8}$$

If we can efficiently calculate $P\left(\boldsymbol{x}_l^{(1)}, \boldsymbol{x}_k^{(2)}\right)$ and $P\left(\boldsymbol{x}_k^{(2)}\right)$, the probability of this query can be easily obtained. Thus, our task in this thesis mainly focus on the exact value of $P\left(\boldsymbol{x}_l^{(1)}, \boldsymbol{x}_k^{(2)}\right)$.

Note that even in the simplest case that every vertex is binary-valued, if we calculate the probability of any given vertex sets directly from the joint distribution induced by a BN, computation time grows exponentially with the number of vertexes in a BN. Such computation is usually prohibitive in exact inference. One direct method for improve the computation is to use *Multi-Linear Functions* (MLFs) [26, 10, 13].

2.2.3 Multi-linear Functions for BNs

An MLF consists of two types of variables: an *indicator variable* $\lambda_{i,k} \in \{0,1\}$ and a *parameter variable* $\theta_{i,j,k}$. $\lambda_{i,k} = 1$ means that vertex X_i takes its k-th value and $\lambda_{i,k} = 0$ otherwise. An MLF contains terms for all instantiations of vertexes:

$$\mathrm{MLF}_V = \sum_{l:v_l \in \mathrm{Dom}(V)} \prod_{i,j,k:x_{i,k} \in v_l, \boldsymbol{\pi}_{i,j} \subset v_l} \lambda_{i,k}\theta_{i,j,k}. \tag{2.9}$$

$\prod_{i,j,k:}$ refers to the multiplication of variables ranging with all values that i, j, k can take. Likewise, $\sum_{l:}$ refers to the summation over variables ranging with all

values that l can take. An MLF represents the joint distribution induced by a BN. The MLF for the example in Fig 2.1 is given by:

$$\begin{aligned}\mathrm{MLF}_V =&\lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\lambda_{4,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1}\theta_{4,1,1}\\ &+\lambda_{1,2}\lambda_{2,1}\lambda_{3,1}\lambda_{4,1}\theta_{1,1,2}\theta_{2,2,1}\theta_{3,2,1}\theta_{4,1,1}\\ &+\lambda_{1,1}\lambda_{2,2}\lambda_{3,1}\lambda_{4,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,1,1}\theta_{4,2,1}\\ &\ldots\\ &+\lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\lambda_{4,2}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,2,2}\theta_{4,4,2}.\end{aligned} \tag{2.10}$$

Computing the probability of an instantiation of any vertices using MLFs is performed by setting all indicators variables consistent with the instantiation to 1 and otherwise to 0. For example, the marginal probability of $x_{4,2}$ can be computed by evaluating this MLF through respectively setting $\lambda_{4,1} \leftarrow 0, \lambda_{4,2} \leftarrow 1$ (other λs are set to 1 and θs are set to the values in CPT). Time of exact inference using this MLF is linear in the numbers of variables in the MLF [12]. Because the number of variables in the MLF grows exponentially with the number of vertexes in the BN, exact inference with a MLF is quite time-consuming. However, if the MLF is factored into a compact arithmetic expression, it is possible to speed up the inference. One way to do factorization using Zero-suppressed BDDs has been proposed in [42].

2.3 ZDD-Based Exact Inference

2.3.1 Zero-Suppressed BDDs

A *Zero-suppressed Binary Decision Diagrams* (ZDD)[39] is a variant of a BDD (Binary Decision Diagram) [1]. A BDD is well known as a data structure to manipulate Boolean functions. ZDDs are much more efficient than BDDs in dealing with combinatorial item sets.

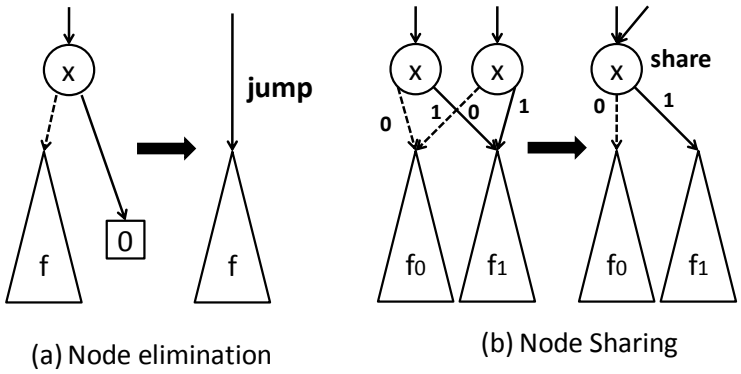


Figure 2.2 ZDD reduction rules

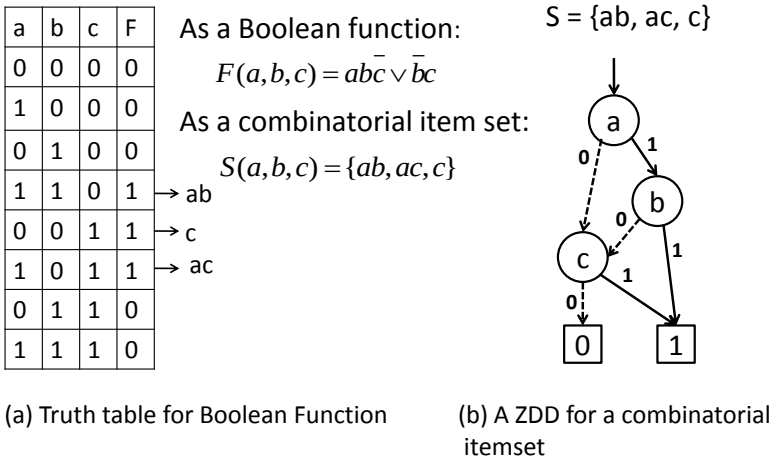
A ZDD consists of two terminal nodes[*1], *0-terminal node* and *1-terminal node*, and decision nodes with exactly two outgoing edges, called *0-edge* and *1-edge*. Each decision node is labeled by a Boolean variable and 1-edge (0-edge) indicates that the variable is true (false).

The reduction is based on the following rules [41] (See Fig 2.2).

- Delete all nodes whose 1-edge directly points to a 0-terminal node, and jump through to the 0-edge's destination, as shown in Fig. 2.2 on the right.
- Sharing equivalent nodes having the same variable and the same pair of child nodes

The ZDDs package proposed by [39] supports a set of various basic logic operations (i.e., AND, OR and XOR) for given a pair of operand ZDDs. These operations take an exponential time for the number of variables in a given combinatorial item set in the worst case. By using cache memory to

[*1] We use the terms "nodes" and "edges" in ZDDs to distinguish with the terms "vertexes" and "arcs" in BNs.



(a) Truth table for Boolean Function

(b) A ZDD for a combinatorial itemset

Figure 2.3 An example of a ZDD

avoid duplicate executions for equivalent subgraphs, ZDDs can be generated and manipulated within a time almost proportional to the size of ZDD itself. By using those inter-ZDD operations, one can construct ZDDs for any given Boolean functions. The details of ZDDs are described in [39].

ZDD rules help to delete the items which never appear in a combinations. Thus, ZDDs are efficient representation of not only Boolean functions but also *combinatorial itemsets* [41]. A combinatorial itemset can be represented as a Boolean space of m input binary variables. For example, the truth table of the Boolean function $F = (ab\bar{c}) \vee (\bar{b}c)$ in 2.3(a) also represents the combinatorial itemset $S = \{ab, ac, c\}$, which is the family of input itemsets that makes F true. An example of ZDD representing S is shown in Fig 2.3(b) [41]. A path in the ZDD from the root node to the 1-terminal node corresponds to an itemset of S. Nodes of irrelevent items (never appeared in this itemset) are automatically deleted from the path.

Given multiple combinatorial itemsets of the same items, isomorphic subgraphs of their ZDDs can be shared under the same fixed input binary vari-

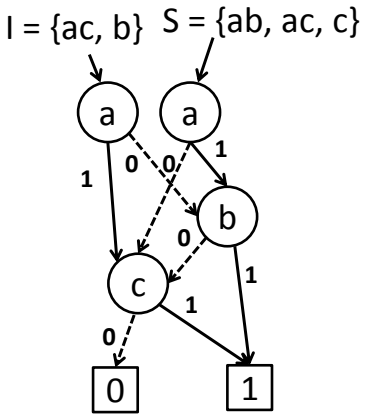


Figure 2.4 An example of sharing ZDD

able order. Fig 2.4 [41] shows an example of the shared ZDDs for combinatorial itemsets I and S. Using node-sharing techniques, one can handle a number of combinatorial itemsets in the time approximately proportional to the compressed ZDD size but not the number of terms of the combinatorial itemset.

2.3.2 Compiling BNs into ZDDs

An MLF is an multi-variant polynomial in the indicator and parameter variables. Since each term of MLF is simply a combination of variables, it can be represented compactly by a ZDD. The conventional ZDD-based method compiles each BN vertex into one MLF based on the MLFs of its parents. Multiplying all the MLFs can get the MLF representing the joint distribution of the given BN equivalent to equation (2.9). For each BN vertex X_i, the MLF is

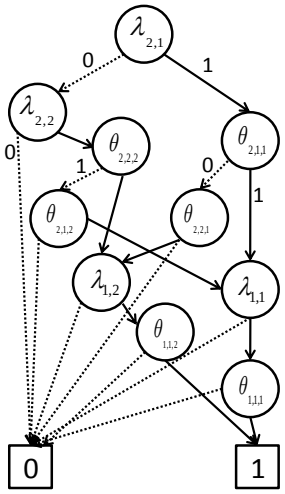


Figure 2.5 A ZDD for MLF_{X_2}.

recursively defined as:

$$\mathrm{MLF}_{X_i} = \sum_{k:x_{i,k}\in \mathrm{Dom}(X_i)} \mathrm{MLF}_{x_{i,k}}, \tag{2.11}$$

$$\mathrm{MLF}_{x_{i,k}} = \lambda_{i,k} \sum_{j:\boldsymbol{\pi}_{i,j}\in \mathrm{Dom}(\boldsymbol{\Pi}_i)} (\theta_{i,j,k} \prod_{x_{i'k'}\in\boldsymbol{\pi}_{i,j}} \mathrm{MLF}_{x_{i'k'}}) \tag{2.12}$$

For the example in Fig 2.1, the MLF at vertex X_1 is:

$$\begin{aligned} \mathrm{MLF}_{X_1} &= \mathrm{MLF}_{x_{1,1}} + \mathrm{MLF}_{x_{1,2}} \\ &= \lambda_{1,1}\theta_{1,1,1} + \lambda_{1,2}\theta_{1,1,2}. \end{aligned} \tag{2.13}$$

Then the MLF for X_2 can be written as:

$$\begin{aligned} \mathrm{MLF}_{X_2} =& \lambda_{2,1}(\theta_{2,1,1}\mathrm{MLF}_{x_{1,1}} + \theta_{2,2,1}\mathrm{MLF}_{x_{1,2}}) \\ &+ \lambda_{2,2}(\theta_{2,1,2}\mathrm{MLF}_{x_{1,1}} + \theta_{2,2,2}\mathrm{MLF}_{x_{1,2}}) \\ =& \lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2} \\ &+ \lambda_{1,2}\lambda_{2,1}\theta_{1,1,2}\theta_{2,2,1} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2}. \end{aligned} \tag{2.14}$$

A ZDD for each MLF is constructed by the multi-valued multiplication algorithm in [42]. The product of MLFs produces all possible combinations

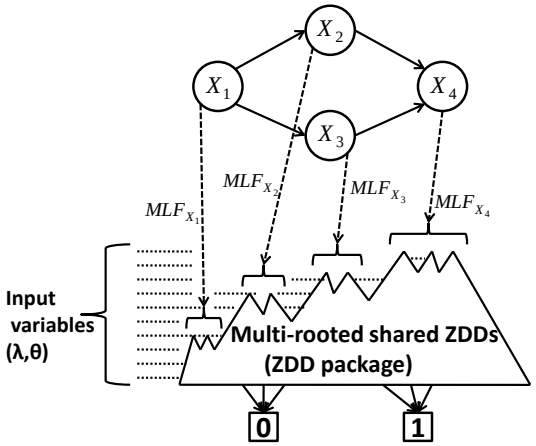


Figure 2.6 ZDD construction

of terms from the respective MLFs. Variables such as $\lambda_{i,k}$ and $\lambda_{i,k'}$ $(k \neq k')$ (variables representing different instance of the same vertex) do not coexist in the same term as they are mutually exclusive. Also, the multi-valued multiplication algorithm with ZDDs operations makes sure that no term can contain the same variable more than once, so instead of $\lambda_{1,1}^2$ for duplicate variables, simply $\lambda_{1,1}$ will appear in the result.

An implicit factored representation of MLF_{X_2} in equation (2.14) with node sharing ZDDs are presented in Fig 2.5. The ZDD for MLF_{X_2} containing only variables relevant to X_2 and its ancestor X_1. In this example, there are four paths from the root node to the 1-terminal node, each of which corresponds to a term of MLF_{X_2}. All MLFs for respective BN vertexes are compactly represent by the shared ZDDs as shown in Fig 2.6 [42].

Comparing with equation (2.6), computing marginal probability for any given vertexes using the conventional ZDD-based method is reduced to the sum of joint distributions over their ancestor vertexes. Irrelevant variables of non-

descendants are out of consideration automatically.

$$P(\boldsymbol{X}) = \sum_{\mathrm{Ances}(\boldsymbol{X})} P(\boldsymbol{X}, \mathrm{Ances}(\boldsymbol{X}))\,. \tag{2.15}$$

Using ZDDs, the exponential size of MLFs for any given BN can be condensed to ZDD size. However, when the number of vertexes in Ances($\boldsymbol{X}$) is too large, the time for compilation and computing $P(\boldsymbol{X}, \mathrm{Ances}(\boldsymbol{X}))$ in equation (2.15) would be extremely time consuming. Therefore, methods to reduce time for both compilation and inference need to be put forward.

Chapter 3

Factorizing ZDDs Based on d-Separation Structures

In this chapter, to condense ZDD size, we represent an improvement of compiling MLFs using ZDDs by combining weak division algorithm with d-separation. In our method, we use the d-separation structure in BN to quickly find a good divisor to factor MLFs into compact representations and we get much more compact MLFs than the conventional ZDD-based method. For example, ZDD size for network of HAILFINDER is reduced about 3 times. For network of MILDEW, ZDD size is reduced about 5 times.

3.1 Introduction

We present an improvement of the conventional ZDD-based method of compiling MLFs into more factored forms based on ZDDs using d-separations. We first introduce the weak division algorithm the most successful and prevalent technique of logic synthesis and optimization and show that if we treat MLFs as logic polynomials, we can use this algorithm to factor MLFs into compressed forms. These operations can be executed in a time almost linear with the ZDD size [38, 41, 40]. Then we explain that for this algorithm, finding a good divisor to factor MLFs is the key to success. Finally, we illustrate that the structure of d-separation used to check some conditional independence in BNs is effective to help us to find a good divisor to execute this factorization.

3.2 Fast Weak Division Algorithm

The weak divison algorithm [21] is the most successful and prevalent technique of logic synthesis and optimization. For optimizing a two-level logic (a form of the Boolean expressions with the AND-OR two level structure), we first generate multi-level logics from it and then apply weak division algorithm to factor the two-level logics. When we determine a intermediate logic, we make a new variable to present it and regard it as a divisor. Then we reduce the other existing logics by factoring them with the divisor. Eventually, we construct a multi-level logic that consists of a number of small two-level logics. The weak division algorithm is executed to computing the common part of quotients for respective items in the divisor. For example, suppose the two

expressions are

$$f = abd + abe + abg + cd + ce + ch, \tag{3.1}$$

and

$$p = ab + c. \tag{3.2}$$

If we write f as:

$$f = ab(d + e + g) + c(d + e + h), \tag{3.3}$$

we factor f by divisor p and the quotient $(f = p)$ can then be computed as:

$$(f/p) = (f/(ab)) \cap (f/c) = (d + e + g) \cap (d + e + h) = d + e \tag{3.4}$$

The remainder $(f\%p)$ is computed using the quotient:

$$(f\%p) = f - p(f/p) = abg + ch. \tag{3.5}$$

Using the quotient and the remainder, we can rewrite f as follows:

$$f = p(p = f) + (f\%p) = pd + pe + abg + ch. \tag{3.6}$$

In this example, equation (3.1) with 15 literals is reduced to 12 literals ($f = pd + pe + abg + ch$ has 9 literals and $p = ab + c$ has 3 literals). Then if we have divisor like $d + e$, we can continue factorization as above to condense the size of f furthermore. Minato [41] has proposed a fast weak division algorithm to refine this algorithm as described in Algorithm 1.

Algorithm 1 procedure(P/Q)

```
1: if Q=1; then
2:    return P;
3: end if
4: if P==0 or P == 1 then
5:    return 0;
6: end if
7: if P == Q; then
8:    return 1;
9: end if
10: R ←cache ("P/Q")
11: if R exists then
12:    return R;
13: end if
14: //the highest variable in Q
15: v ← Q.top;
16: (P0, P1) ← factors of P by v;
17: (Q0, Q1) ← factors of Q by v (Q1 ≠ 0);
18: R ← P1/Q1;
19: if R ≠ 0 and Q ≠ 0 then
20:    R ← R ∩ P0/Q0;
21: end if
22: cache ("P/Q")← R;
23: return R
```

They implicitly represent logics using ZDDs and manipulate them using ZDD operations. The fast weak division algorithm is computed in a time almost proportional to the number of nodes in ZDDs, which are usually much smaller than the number of literals in logics, and is much faster than conventional methods [41]. Thus if we have a proper divisor, we can quickly condense the size of a given polynomial by using this fast weak division algorithm. So we consider to use this approach to factor a given MLF and the main problem now is how to find a proper divisor for a MLF.

3.2.1 Problem in Factoring MLFs

For an MLF of a given BN, if we consider it as a polynomial, we can extract repeatedly appeared variables by using the fast weak division algorithm to condense its size. Here we use MLF_{X_2} in Fig. 2.1 as an example.

$$\begin{aligned}\mathrm{MLF}_{X_2} = {} & \lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2} \\ & + \lambda_{1,2}\lambda_{2,1}\theta_{1,1,2}\theta_{2,2,1} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2}.\end{aligned} \tag{3.7}$$

If we use $\lambda_{2,1}\theta_{2,1,1} + \lambda_{2,2}\theta_{2,2|1,1}$ as a *divisor* P, according to the algorithm, MLF_{X_2} can be factored as follows:

$$\begin{aligned}& \mathrm{MLF}_{X_2} / (\lambda_{2,1}\theta_{2,1,1} + \lambda_{2,2}\theta_{2,1,2}) \\ & = (\mathrm{MLF}_{X_2} / (\lambda_{2,1}\theta_{2,1,1}) \cap (\mathrm{MLF}_{X_2} / \lambda_{2,2}\theta_{2,1,2}) \\ & = (\lambda_{1,1}\theta_{1,1,1}) \cap (\lambda_{1,1}\theta_{1,1,1}) \\ & = \lambda_{1,1}\theta_{1,1,1}\end{aligned} \tag{3.8}$$

Finally, MLF_{X_2} can be rewritten as:

$$\mathrm{MLF}_{X_2} = \lambda_{1,1}\theta_{1,1,1} * P + \lambda_{1,2}\lambda_{2,1}\theta_{1,1,2}\theta_{2,2,1} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2} \tag{3.9}$$

However, if we factor MLF_{X_2} using factor $P = \lambda_{1,1}\theta_{1,1,1} + \lambda_{1,2}\theta_{1,1,2}$, the quotient will be the empty set so MLF_{X_2} can not be rewritten by *divisor* P. Therefore, the quality of the results of this algorithm greatly depends on the choice of *divisors*.

3.2.2 Divisor Extraction Based on BN Vertexes

The MLF of a vertex in a given BN is based on its parents nodes. For the vertex X_2 in Fig. 2.1, MLF_{X_2} contains information about vertex X_1. Here we

refer to this information with parameters $x_{1,1}$ and $x_{1,2}$. Also, if the number of parameters of vertex X_1 and X_2 are given, we can forecast the size of MLF_{X_2} and the frequency of characters λ and θ. Therefore, we consider factoring an MLF of a vertex with the MLF of its parents directly. But, this fails when we implement $\mathrm{MLF}_{X_2}/\mathrm{MLF}_{X_1}$. We give the details next.

$$
\begin{aligned}
&\mathrm{MLF}_{X_2}/\mathrm{MLF}_{X_1} \\
&= \mathrm{MLF}_{X_2}/(\mathrm{MLF}_{x_{1,1}} + \mathrm{MLF}_{x_{1,2}}) \\
&= (\mathrm{MLF}_{X_2}/(\mathrm{MLF}_{x_{1,1}}) \cap (\mathrm{MLF}_{X_2}/(\mathrm{MLF}_{x_{1,2}}) \\
&= \{(\lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2})/\lambda_{1,1}\theta_{1,1,1}\} \cap \\
&\quad \{(\lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2})/\lambda_{1,2}\theta_{1,1,2}\} \\
&= (\lambda_{2,1}\theta_{2,1,1} + \lambda_{2,2}\theta_{2,1,2}) \cap (\lambda_{2,1}\theta_{2,2,1} + \lambda_{2,2}\theta_{2,2,2}) \\
&= \emptyset.
\end{aligned}
\tag{3.10}
$$

We refer to the division of $\mathrm{MLF}_{X_2}/\mathrm{MLF}_{X_1}$ as blotting out information about vertex X_1. Why we get the empty set is that though we try to blot out $x_{1,1}$ by $\mathrm{MLF}_{X_2}/\mathrm{MLF}_{x_{1,1}}$, $x_{1,1}$ is still left in $\theta_{2,1,1}$ and $\theta_{2,1,2}$. The same applies to $x_{1,2}$. When we intersect the quotients, which are obtained by factoring MLF_{X_2} with $\mathrm{MLF}_{x_{1,1}}$ and $\mathrm{MLF}_{x_{1,2}}$, $x_{1,1}$ and $x_{1,2}$ are contrary, hence we obtain the empty set. But, if we omit the intersection, which means we perform the factorization as $\mathrm{MLF}_{X_2}/\mathrm{MLF}_{x_{1,1}}$, $\mathrm{MLF}_{X_2}/\mathrm{MLF}_{x_{1,2}}$, MLF_{X_2} can be rewritten as:

$$
\mathrm{MLF}_{X_2} = \mathrm{MLF}_{x_{1,1}}(\lambda_{2,1}\theta_{2,1,1} + \lambda_{2,2}\theta_{2,1,2}) + \mathrm{MLF}_{x_{1,2}}(\lambda_{2,1}\theta_{2,2,1} + \lambda_{2,2}\theta_{2,2,2}) \tag{3.11}
$$

However, even this works only in the case of a vertex which has only one parent vertex like vertex X_2. If it has more than one parent vertex, for example, vertex X_3 in Fig. 2.1, the representations of $\mathrm{MLF}_{x_{2,1}}$, $\mathrm{MLF}_{x_{2,2}}$ are

not capable of factoring MLF_{X_3}.

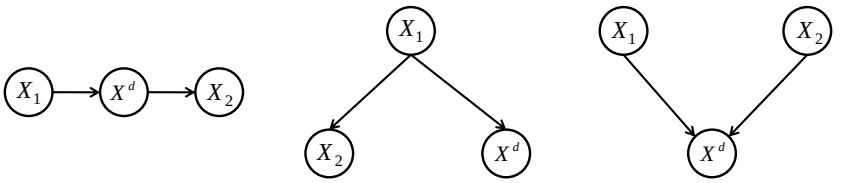
$$\begin{aligned}
&\mathrm{MLF}_{X_4}/\mathrm{MLF}_{x_{2,1}} \\
&= (\lambda_{3,1}\lambda_{4,1}\theta_{3,1,1}\theta_{4,1,1} + \cdots + \lambda_{3,2}\lambda_{4,3}\theta_{3,1,1}\theta_{4,2,3}) \cap \\
&\quad (\lambda_{3,1}\lambda_{4,1}\theta_{3,2,1}\theta_{4,1,1} + \cdots + \lambda_{3,2}\lambda_{4,3}\theta_{3,2,2}\theta_{4,2,3}) \\
&= \emptyset
\end{aligned} \tag{3.12}$$

The reason we get the empty set is since MLF_{X_2} is based on vertex X_1, when we try to blot out the information about $\mathrm{MLF}_{x_{2,1}}$ by $\mathrm{MLF}_{X_3}/\mathrm{MLF}_{x_{2,1}}$, we are also blotting out information about $x_{1,1}$ and $x_{1,2}$ contained in MLF_{X_4}. The blotting out is inadequate because for $x_{1,1}$ and $x_{1,2}$ are also contained in MLF_{X_3} and they contradict to each other when we intersect the quotients. Thus, this motivates us to find a vertex set that can separate vertex X_1 and vertex X_4 as independent vertexes so that after we factoring MLF_{X_4}, the information about vertex X_1 can be cleared up thoroughly. We propose an idea of factoring MLFs using the combinations of variables of d-separation vertex sets so solve the problems mentioned above.

3.3 Divisor Extraction Based on d-Separations

3.3.1 d-Separations

The structure of d-separation is used to check conditional independence between variables in BNs. It can be presented as three graph patterns [28] as shown in Fig 3.1. The d-separation has an important property that if we substitute the observed values to the d-separation vertexes, vertexes in both sides cut by the d-separation become independent. The approach we propose is based on d-separation of serial pattern.



(a) serial pattern (b) diverging pattern (c) converging pattern

Figure 3.1 An example of d-separation.

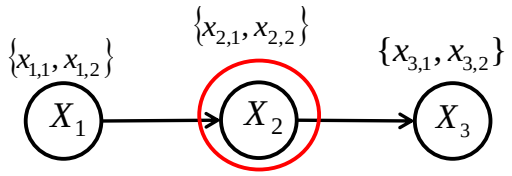


Figure 3.2 An example of one-vertex d-separation.

For a given MLF to be factored which is represented by ZDD, first we try to find suitable d-separation vertex set and multiply their MLFs. Then we consider the result of multiplication as a divisor to factor the MLF using fast weak algorithm. We first try from the most simple d-separation which consists of only one vertex (one-vertex d-separation). Since the one d-separations not always exist in BNs, we use d-separations which consist of two or three vertexes (multi-vertex d-separations). However, the multi-vertex d-separations are found manually.

3.3.2 Divisor Selection by One-vertex d-Separations

For an MLF to be factored by one-vertex d-separation, we use the MLF of every variable of this vertex as its divisor. For the example in Fig. 3.2, the

MLF of vertex X_3 is:

$$\begin{aligned}\mathrm{MLF}_{X_3} = {} & \lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \lambda_{1,1}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2} \\ & + \lambda_{1,1}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,1,2} + \lambda_{1,1}\lambda_{2,2}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,2,2} \\ & + \lambda_{1,2}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,2}\theta_{2,2,1}\theta_{3,1,1} + \lambda_{1,2}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,2}\theta_{2,2,1}\theta_{3,1,2} \\ & + \lambda_{1,2}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,1,2} + \lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,2,2}\end{aligned} \tag{3.13}$$

The MLFs of variables of vertex X_2 are:

$$\begin{aligned}\mathrm{MLF}_{x_{2,1}} &= \lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,2}\lambda_{2,1}\theta_{1,1,2}\theta_{2,2,1} \\ \mathrm{MLF}_{x_{2,2}} &= \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2}\end{aligned} \tag{3.14}$$

We can factor MLF_{X_3} using $\mathrm{MLF}_{x_{2,2}}$and $\mathrm{MLF}_{x_{2,1}}$ because X_2 is the single vertex that separates vertex X_1 and X_3.

$$\begin{aligned}\mathrm{MLF}_{X_3}/\mathrm{MLF}_{x_{2,1}} =& (\lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \ldots \lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\theta_{x_{2,2}}\theta_{2,2,2}\theta_{3,2,2}) \\ & /(\lambda_{1,1}\lambda_{2,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{1,2}\lambda_{2,1}\theta_{1,1,2}\theta_{2,2,1}) \\ =& (\lambda_{3,1}\theta_{3,1,1} + \lambda_{3,2}\theta_{3,1,2}) \cap (\lambda_{3,1}\theta_{3,1,1} + \lambda_{3,2}\theta_{3,1,2}) \\ =& \lambda_{3,1}\theta_{3,1,1} + \lambda_{3,2}\theta_{3,1,2}\end{aligned} \tag{3.15}$$

$$\begin{aligned}\mathrm{MLF}_{X_3}/\mathrm{MLF}_{x_{2,2}} =& (\lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \ldots \lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\theta_{x_{2,2}}\theta_{2,2,2}\theta_{3,2,2}) \\ & /(\lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2}) \\ =& (\lambda_{3,1}\theta_{3,1,2} + \lambda_{3,2}\theta_{3,2,2}) \cap (\lambda_{3,1}\theta_{3,1,2} + \lambda_{3,2}\theta_{3,2,2}) \\ =& \lambda_{3,1}\theta_{3,1,2} + \lambda_{3,2}\theta_{3,2,2}\end{aligned} \tag{3.16}$$

Finally, we rewrite MLF_{X_3} as follows:

$$\mathrm{MLF}_{X_3} = \mathrm{MLF}_{x_{2,1}}(\lambda_{3,1}\theta_{3,1,1} + \lambda_{3,2}\theta_{3,1,2}) + \mathrm{MLF}_{x_{2,2}}(\lambda_{3,1}\theta_{3,1,2} + \lambda_{3,2}\theta_{3,2,2}) \tag{3.17}$$

3.3.3 Divisor Selection by Multi-vertex d-Separations

We use the BN of Fig 2.1 to show how to perform the factorization based on multi-vertex d-separation. For vertex X_1 and X_4, $\{X_2, X_3\}$ is the d-separation vertex set that separates them as independent vertexes. Since both vertex X_2 and X_3 have two values, there are four combinations of of their instantiations $\{x_{2,1}x_{3,1}, x_{2,1}x_{3,2}, x_{2,2}x_{3,1}, x_{2,2}x_{3,2}\}$. According to [42], we multiply their MLFs as follows. There are two terms in each of these MLFs, so the number of terms after multiplication should be $2 * 2 = 4$. But since the parameters λ are eliminated if they contradict each other, only two of the four terms are left. Following shows the details of the multiplication.

$$\begin{aligned}
&\mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}} \\
&= (\lambda_{2,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{2,1}\lambda_{1,2}\theta_{1,1,2}\theta_{2,2,1}) \\
&\quad (\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{3,1,1} + \lambda_{3,1}\lambda_{1,2}\theta_{1,1,2}\theta_{3,2,1}) \\
&= \lambda_{2,1}\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \lambda_{2,1}\lambda_{3,1}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,1}\theta_{3,2,1}.
\end{aligned} \tag{3.18}$$

$$\begin{aligned}
&\mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,2}} \\
&= (\lambda_{2,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1} + \lambda_{2,1}\lambda_{1,2}\theta_{1,1,2}\theta_{2,2,1}) \\
&\quad (\lambda_{3,2}\lambda_{1,1}\theta_{1,1,1}\theta_{3,1,2} + \lambda_{3,2}\lambda_{1,2}\theta_{1,1,2}\theta_{3,2,2}) \\
&= \lambda_{2,1}\lambda_{3,2}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2} + \lambda_{2,1}\lambda_{3,2}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,1}\theta_{3,2,2}.
\end{aligned} \tag{3.19}$$

$$\begin{aligned}
&\mathrm{MLF}_{x_{2,2}}\mathrm{MLF}_{x_{3,1}} \\
&= (\lambda_{2,2}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,2} + \lambda_{2,2}\lambda_{1,2}\theta_{1,1,2}\theta_{2,2,2}) \\
&\quad (\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{3,1,1} + \lambda_{3,1}\lambda_{1,2}\theta_{1,1,2}\theta_{3,2,1}) \\
&= \lambda_{2,2}\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,1,1} + \lambda_{2,2}\lambda_{3,1}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,2}\theta_{3,2,1}.
\end{aligned} \tag{3.20}$$

$$\begin{aligned}
&\mathrm{MLF}_{x_{2,2}}\mathrm{MLF}_{x_{3,2}} \\
&= (\lambda_{2,2}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,2} + \lambda_{2,2}\lambda_{1,2}\theta_{1,1,2}\theta_{2,2,2}) \\
&\quad (\lambda_{3,2}\lambda_{1,1}\theta_{1,1,1}\theta_{3,1,2} + \lambda_{3,2}\lambda_{1,2}\theta_{1,1,2}\theta_{3,2,2}) \\
&= \lambda_{2,2}\lambda_{3,2}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,1,2} + \lambda_{2,2}\lambda_{3,2}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,2}\theta_{3,2,2}.
\end{aligned} \tag{3.21}$$

After these multiplication, we factor the MLF_{X_4} with the four combinations respectively. We give an example of $\mathrm{MLF}_{X_4}/\mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}}$ in details.

$$\begin{aligned}
&\mathrm{MLF}_{X_4}/\mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}} \\
&= \mathrm{MLF}_{X_4}/(\lambda_{2,1}\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \\
&\qquad \lambda_{2,1}\lambda_{3,1}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,1}\theta_{3,2,1}) \\
&= \mathrm{MLF}_{X_4}/(\lambda_{2,1}\lambda_{3,1}\lambda_{1,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1}) \cap \\
&\quad \mathrm{MLF}_{X_4}/(\lambda_{2,1}\lambda_{3,1}\lambda_{1,2}\theta_{1,1,1}\theta_{2,2,1}\theta_{3,2,1}) \\
&= \lambda_{4,1}\theta_{4,1,1} + \lambda_{4,2}\theta_{4,1,2} + \lambda_{4,3}\theta_{4,1,3}.
\end{aligned} \tag{3.22}$$

Finally, we can rewrite MLF_{X_4} as follows:

$$\begin{aligned}
\mathrm{MLF}_{X_4} =& \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}}(\lambda_{4,1}\theta_{4,1,1} + \lambda_{4,2}\theta_{4,1,2} + \lambda_{4,3}\theta_{4,1,3}) + \\
& \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,2}}(\lambda_{4,1}\theta_{4,3,1} + \lambda_{4,2}\theta_{4,2,2} + \lambda_{4,3}\theta_{4,3,3}.) + \\
& \mathrm{MLF}_{x_{2,2}}\mathrm{MLF}_{x_{3,1}}(\lambda_{4,1}\theta_{4,2,1} + \lambda_{4,2}\theta_{4,3,2} + \lambda_{4,3}\theta_{4,2,3}.) + \\
& \mathrm{MLF}_{x_{2,2}}\mathrm{MLF}_{x_{3,2}}(\lambda_{4,1}\theta_{4,4,1} + \lambda_{4,2}\theta_{4,4,2} + \lambda_{4,3}\theta_{4,2,3}.).
\end{aligned} \tag{3.23}$$

3.4 Experiments and Results

We implement our experiments on the platform of a Intel Core Quad CPU Q9550@2.83GHz * 4 PC with Ubuntu 12.04LTS and 3.8GiB of main memory. We manipulate up to 40,000,000 nodes of ZDDs. We use data set of BN Benchmark [23] ALARM, HAILFINDER, INSURANCE and MILDEW to implement our experiment.

We try to choose the vertexes which have the biggest size of ZDD in ALARM and HAILFINDER. For INSURANCE and MILDEW. considering

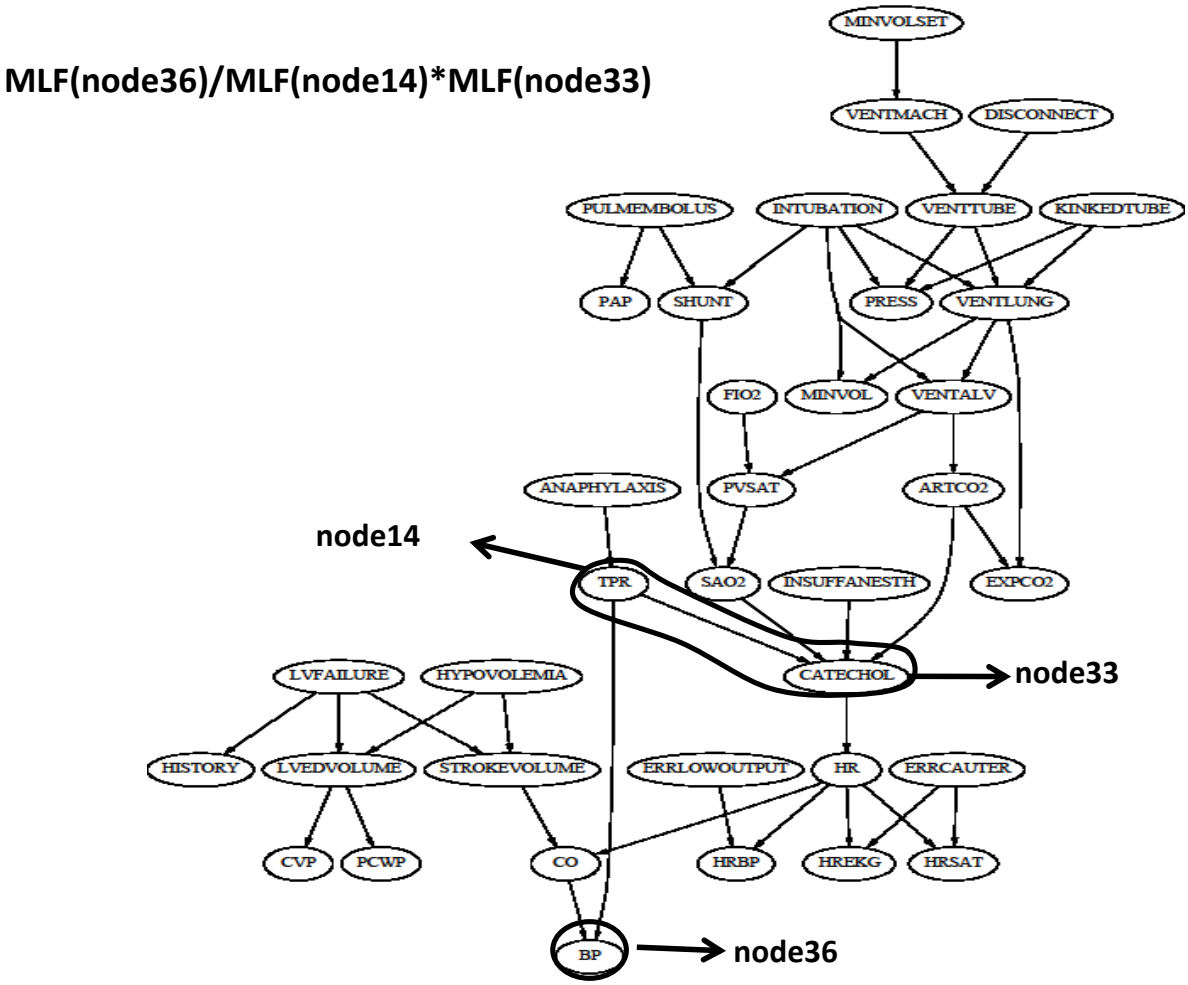


Figure 3.3 The example of ALARM36(n14n33)

time consumption, we do not use the vertexes with biggest ZDD size but choose the vertexes which seem to have suitable d-separation vertex sets.

To compare with our method, we use a simple criterion to extract divisors from MLFs, that is abstracting variables that appear more than twice in the MLFs. The experiment results are shown in Table 3.2. The first column shows the vertex with its number in the BN. For example, ALARM36 means

Table 3.1 Original MLFs and ZDDs before factorization.

Dataset	BN specifications			Before factorization			
	BN vertexes	Indi-cators	Para-meters	ZDD size	Total terms	Total literals	Generating ZDD
ALARM36	37	105	187	4,551	>100b*	>500b	0.647s
HAILFINDER43	56	223	835	73,700	>2b	>210m*	45.902s
INSURANCE5	27	–	–	6,182	628,992	>10m	11.768s
INSURANCE14	27	–	–	56,490	>70m	>2b	11.603s
MILDEW14	35	616	6,709	80,248	>2b	>2b	946.35s
MILDEW20	35	616	6,709	107,828	>2b	>2b	947.73s

*'b' means billion and 'm' means million

Table 3.2 Factorization without d-separation.

Dataset and vertex ID	Factorization without d-separation			
	ZDD size	Total terms	Total literals	Time for factorization
ALARM36	6,784	3,500	13,512	1969.77ss
HAILFINDER43	overflow	–	–	–
INSURANCE5	5,256	3,026	10,483	2.169s
INSURANCE14	overflow	–	–	–
MILDEW14	overflow	–	–	–
MILDEW20	overflow	–	–	–

Table 3.3 Factorization with d-separation.

Dataset and vertex ID and d-separation vertex set	After factorization based on d-separation			
	ZDD size	Total terms	Total literals	Factorization
ALARM36(n14n33)	5,178	2662	9,796	51.425s
ALARM36(n20n32)	4,247	2,423	8,776	4.825s
ALARMN36(n14n33/n20n32)	4,133	2,342	8,422	3.427s
HAILFINDER43(n14n20)	24,353	>2G	>2G	391.648s
HAILFINDER43(n14n20/n4n12)	22,833	14,512	50,113	303.641s
INSURANCE5(n4n8n9)	1,985	1,359	3,924	0.24s
INSURANCE14(n2n3n9)	34,222	22,088	72,882	2427.79s
MILDEW14(n11n12)	14,727	11,785	33,355	619.391s
MILDEW20(n17n18)	22,356	16,784	51,242	754.577s
MILDEW20(n17n24)	overflow	–	–	–

we factor the MLF of vertex 36 in ALARM. The other columns show the ZDD specifications after factoring and time consumption with the simple divisor extracting algorithm to show the upper limits of consumption of calculation and inference based on MLFs without any compression techniques.

For the vertexes we use in Table 3.2, we use algorithm to find one-vertex d-separation and manually find multi-vertex d-separations and use our method to factor the MLFs. The experiment results are shown in Table 3.3. The first column in this table lists vertexes with numbers and d-separation sets we choose in BN. For instance, ALARM36(n14n33) means factoring the MLF of vertex 36 with d-separation set MLFs of vertex 14 and vertex 33 (Fig. 3.3). According to Table 3.3, we could always achieve quite smaller ZDDs and condense MLFs quite efficiently using our method comparing to Table 3.1 and Table 3.2. However, we get a bigger ZDDs due to the factorization in 'ALARM36'. This is because the number of newly introduces variables to represent *divisors* are more than the reduction of ZDD nodes. Since the size of ZDD is depended on the structure of BN itself and the probability of every instance, we can not precisely tell how much we can condense MLFs. We hope to find the d-separation vertex sets with minimum number of combinations of variables to reduce the newly introduces variables. This is the reason why we just use one or two nodes d-separations but not more. For ALARM36 and HAILFINDER43, we get different results with different d-separation sets. That is to say, if we can find suitable d-separations, we will further condense ZDD size which is important because it determines the time and space requirements for online inference which is linear in ZDD size.

3.5 Summary

We represented an improvement of compiling MLFs using ZDDs by combining weak division algorithm with d-separations. In our method, we use the d-separation structures in BNs to quickly find good divisors to factor MLFs into compact representations and we get much more compact MLFs than the

conventional ZDD-based method [42].

In our method, we first generate ZDDs for the whole given BN, and then factor the ZDDs using d-separation vertexes. The process of factoring costs too much time and sometime the ZDD for a BN is too large to factor, such as MILDEW20(n17n24) in Table 3.3, though we could find proper d-separation nodes. To improve the factorization, we expect to develop simple and fast heuristic algorithms to find d-separation sets as divisors instead of finding d-separation sets manually. For example, utilizing the structure of junction tree. Also we want to consider not to generate the ZDD for the whole network but just generate the ZDD for the newfound divisor, we may avoid the time consumption and get quite compact ZDDs. For example, ALARM36(n14n33/n20n32) shows that if we first factor vertex 36 with set of vertex 14 and 33, then factor the result with set of vertex 20 and 32, we can get a smaller ZDD using this two-level d-separations. It gives us a hint that in the process of generating the ZDD from the MLF of ALARM, we can first generate a ZDD for *divisor* of vertex 20 and 32, then using this ZDD to generate a ZDD of divisor vertex 14 and 33. Finally, we use the two ZDDs to generate the ZDD of the whole network so that we avoid the time consumption of factoring and also reduce the ZDD. These expectations help bring about the ideas in the next two chapters.

Chapter 4

Fast Message Passing Algorithm Using ZDD-Based Local Structure Compilation

We have shown a method of condensing ZDDs by factorizing large ZDDs into small one. However, this method is limited when a BN is too large to compile into ZDDs. Naturally, we think of compiling some parts of BNs instead of the whole BNs into ZDDs. The junction tree is a factorized form of a BN. It decomposes a BN into a tree structure and message passing algorithm [48], the most popular and prevalent algorithm for exact inference of BNs is carried out on this tree. We propose to accelerate this algorithm by compiling a junction tree into ZDDs and perform message passing algorithm on ZDDs. Experiments show that through our compilation, inference are much faster than the original junction tree algorithm. For the

networks of ALARM and INSURANCE, inference is accelerated about 10 times than the original junction algorithm.

4.1 Introduction

A *junction tree* [33], also known as *jointree*, one of the most essential tree structure in exact inference of BNs, whose nodes and edges are labeled with subsets of variables in a BN. By performing message passing algorithm on junction trees, the computation complexity of inference is bound to *treewidth* of a junction tree [48, 46]. Given a junction tree with treewidth w, all marginal probabilities of BN vertexes can be computed in time and space exponential in w. To generate a junction tree, one needs to triangulate the moral graph of the given BN. Finding the best order to triangulate a moral graph will lead to a junction tree with minimum treewidth. However, this is known as an *NP hard* problem. Constructing a junction tree with minimal width is a hot topic for junction tree based approaches [47, 34, 35]. *Local structures* [33] in the forms of variables sharing the same values or repetitive local computations are known as the breakthrough of the treewidth barrier. The ability to exploit local structures to avoid redundant calculations are known as a significant impact on exact inference [18].

In this chapter, we discuss how to accelerate the essential inference algorithm, message passing algorithm by exploiting local structures with ZDDs. We directly compile a junction tree into ZDDs. We introduce *message variables* whose values dynamically change according to different messages so that we can perform the message passing algorithm on ZDDs. We will show that

through the compilation, we can perform exact inference much more efficiently than the conventional message passing algorithm.

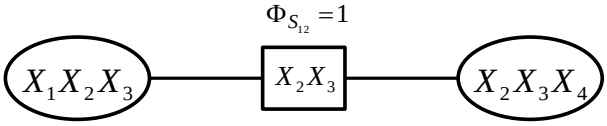
4.2 Junction Tree and Message Passing

A *junction tree* $T = \langle N, A \rangle$ for BN B is a tree structure, where $N = \{1, 2, ..., m\}$ is a set of nodes and $A \subset N \times N$ is a set of undirected edges [*1] (a, b) where $a, b \in N$, $a \neq b$. Each node and each edge is labeled by a subset of variables V: $C_a \subseteq V$ denotes the label of node $a \in N$ that is called a *cluster*, and $S_{ab} \triangleq C_a \cap C_b$ denotes the label of edge $(a, b) \in A$ that is called a *separator*. We use $c_{a,m}$ to denote the m-th instantiation of C_a and $s_{ab.\ell}$ to denote the ℓ-th instantiation of S_{ab}. The size for node C_a represented as $|C_a|$ is the number of variables in C_a. Thus the treewidth of a T equals the size of the largest node. We here define a *family* $F_a \triangleq \{X_i \in V \mid X_i \cup \Pi_i \subseteq C_a\}$. Then, if T is a junction tree, clusters in T satisfies the following properties:

- *Running intersection*: $\forall a, b \in N, \forall c \in N$ on the unique path between a and b, $C_a \cap C_b \subseteq C_c$.
- *Family preserving*: $\forall X_i \in V, \exists a \in N, X_i \in F_a$.

In general, X_i can belong to multiple families. For each X_i, here we choose one family F_a s.t. $X_i \in F_a$ in arbitrary manner and say that X_i is *assigned* to C_a. We use $SN(C_a)$ to denote the set of all variables that are assigned to C_a. Without loss of generality, we assume that T is rooted and use r to denote the root node of T. We use $d(a)$ to denote the depth of a that

[*1] To distinguish with the vertexes and arcs in $B = \langle G, \Theta \rangle$, we use nodes and edges in a junction tree.



$$\Phi_{C_1} = P(X_1)P(X_2 \mid X_1)P(X_3 \mid X_2) \qquad \Phi_{C_2} = P(X_4|X_2,X_3)$$

Figure 4.1 A junction tree for the BN in Fig 2.1.

is the distance from r, and use $N_d \triangleq \{a \in N \mid d(a) = d\}$ and $d(T) \triangleq \max_{a \in N} d(a)$. We use $par(a) \in N$ and $child(a) \subset N$ to denote the parent and the set of children of a in T, respectively.

Message passing algorithm is an algorithm that can compute all marginal probabilities $P(X_i)$ $(X_i \in V)$ on a BN B using junction tree T [33]. In the message passing procedure, a *potential*, which is a function over a set of variables, is recursively updated by *messages* that are computed from the other potentials. We define the potential over C_a and S_{ab} as $\Phi_{C_a}(C_a)$ and $\Phi_{S_{ab}}(S_{ab})$, respectively. We use $\phi_{c_{a,m}} \triangleq \Phi_{C_a}(c_{a,m})$ and $\phi_{s_{ab,\ell}} \triangleq \Phi_{S_{ab,\ell}}(s_{ab,\ell})$, respectively. We use Φ to denote the set of all potentials $\Phi_{C_a}(C_a)$ and $\Phi_{S_{ab}}(S_{ab})$. For the sake of simplicity, we omit the arguments of potentials afterward. The message passing is summed up in Algorithm 2.

When the message passing ends, these potentials satisfy the following equations:

$$\Phi_{C_a} = P(C_a), \quad \Phi_{S_{ab}} = P(S_{ab}), \quad P(V) = \frac{\prod_{a \in N} \Phi_{C_a}}{\prod_{(a,b) \in A} \Phi_{S_{a,b}}}. \tag{4.1}$$

Algorithm 2 MessagePassing(T)

// Initialize
$\phi_{c_{a,m}} \leftarrow \prod_{X_i \in SN(C_a)} P(x_{i,k} \mid \pi_{i,j})$ for all $a \in N$ and m, where $x_{i,k} \cup \pi_{i,j} \subseteq c_{a,m}$
$\phi_{s_{ab,\ell}} \leftarrow 1$ for all $(a,b) \in A$ and ℓ
// Collect messages : child to parent
for $d = d(T) - 1$ to 0 **do**
  for all $b \in N_d$ **do**
    for all $a \in child(b)$ **do**
      Pass$(a, b, \Phi_{C_a}, \Phi_{C_b}, \Phi_{S_{ab}})$
    end for
  end for
end for
// Distribute messages : parent to child
for $d = 1$ to $d(T)$ **do**
  for all $b \in N_d$ **do**
    for all $a \in par(b)$ **do**
      Pass$(a, b, \Phi_{C_a}, \Phi_{C_b}, \Phi_{S_{ab}})$
    end for
  end for
end for
return Φ

Algorithm 3 Pass$(a, b, \Phi_{C_a}, \Phi_{C_b}, \Phi_{S_{ab}})$

$\phi^{\text{old}}_{s_{ab,\ell}} \leftarrow \phi_{s_{ab,\ell}}$ for all ℓ
$\phi_{s_{ab,\ell}} \leftarrow \sum_{c_{a,m} \setminus s_{ab,\ell}} \phi_{c_{a,m}}$, for all ℓ and m where $s_{ab,l} \subset c_{a,m}$
$\phi_{c_{b,m}} \leftarrow \phi_{c_{b,m}} \frac{\phi_{s_{ab,\ell}}}{\phi^{old}_{s_{ab,\ell}}}$ for all ℓ and m where $s_{ab,l} \subset c_{b,m}$

Once we finish the computation for $P(C_a)$, we can easily get $P(X_i)$ ($X_i \in C_a$) by marginalizing out all $X_j \in C_a \setminus \{X_i\}$ from $P(C_a)$ as follows:

$$P(X_i) = \sum_{C_a \setminus \{X_i\}} P(C_a). \tag{4.2}$$

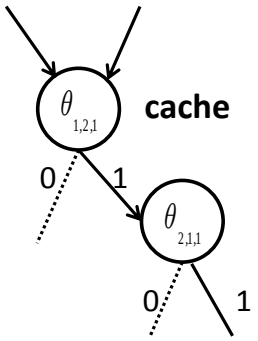


Figure 4.2 An example of local computation.

4.3 Local Structures

Though message passing on junction trees is an efficient method, for each cluster in a junction tree, we need to prepare a table for the potential whose size is exponential to the size of the cluster. Operations on these tables incur in considerable costs in message passing. If we can exploit all the local structures, in the form of equal parameters and local computation, during message passing, the algorithm can be significantly accelerated.

For the example CPTs in Fig 2.1, there are 18 parameters in total in the CPTs and yet only 8 of them are distinct. Compact representations for these CPTs are expected. Also while calculating different probabilities who are sharing the same local computation, for example calculating $P(x_{1,2}, x_{2,1}, x_{3,1}) = \theta_{1,2,1}\theta_{2,1,1}\theta_{3,1,1}$ and $P(x_{1,2}, x_{2,1}, x_{3,2}) = \theta_{1,2,1}\theta_{2,1,1}\theta_{3,2,1}$, multiplication for $\theta_{1,2,1}\theta_{2,1,1}$ are repetitive. If we can cache local computations in advance, there is no need to calculate it again while evaluating a new query which shares the same local computations with queries already calculated.

4.4 Fast Message Passing Algorithm

In this section, we introduce our fast message passing algorithm using ZDDs. We first convert all updates of potentials of naive message passing into MLFs. Then, we compile them into ZDDs in the same manner as the conventional ZDD-based method [42]. First, we define MLFs that we use in our method, then we explain how we can simulate the message passing using those MLFs.

4.4.1 MLFs for Message Passing

Before generating the ZDD, we need to convert potentials in junction tree into MLFs. While converting these components into MLFs, we introduce three types of variables: indicator variable $\lambda_{i,k}$; parameter variable $\theta_{i,j,k}$, with the same definition in Section 2; and *message variables* $\beta^{a}_{s_{ab,\ell}}$, $\beta^{b}_{s_{ab,\ell}}$ for instantiation $s_{ab,\ell}$.

In our method, we need to generate MLFs for $P(X_i \mid \Pi_i)$, $\Phi_{S_{ab}}$ and Φ_{C_a}, for each $X_i \in V, (a,b) \in A, a \in N$. We use MLF_{X_i}, $\mathrm{MLF}_{S_{ab}}$, MLF_{C_a} to represent MLFs for them respectively. We define the corresponding MLFs for these components as follows:

- For all $X_i \in V$, we define

$$\mathrm{MLF}_{X_i} \triangleq \sum_{j,k} \lambda_{i,k} \Lambda_{i,j} \theta_{i,j,k}, \qquad \Lambda_{i,j} \triangleq \prod_{x_{i',k'} \in \pi_{i,j}} \lambda_{i',k'}. \tag{4.3}$$

- For all $(a, b) \in A$, define

$$\mathrm{MLF}^a_{S_{ab}} \triangleq \sum_l \Lambda_{ab,\ell} \beta^a_{s_{ab,\ell}}, \quad \mathrm{MLF}^b_{S_{ab}} \triangleq \sum_l \Lambda_{ab,\ell} \beta^b_{s_{ab,\ell}}, \quad \Lambda_{ab,\ell} \triangleq \prod_{x_{i,k} \in s_{ab,\ell}} \lambda_{i,k}. \tag{4.4}$$

- For all $a \in N$, using the above MLF_{X_i} and $\mathrm{MLF}^a_{S_{ab}}$, we define

$$\mathrm{MLF}_{C_a} \triangleq \prod_{X_i \in SN(C_a)} \mathrm{MLF}_{X_i} \prod_{(a,b) \in A} \mathrm{MLF}^a_{S_{ab}}. \tag{4.5}$$

Using the multi-valued multiplication algorithm [42] in ZDD operations, the product of these MLFs produces all possible combinations of their terms. Variables such as $\lambda_{i,k}$ and $\lambda_{i,k+1}$ (variables representing different instances of the same variable) does not coexist in the same term as they are mutually exclusive. Also, union operations of ZDDs make sure that no term can contain the same variable more than once, so instead of $\lambda^2_{1,1}$ for duplicate variables, simply $\lambda_{1,1}$ will appear in the result.

For example the junction tree in Fig 4.1, first we generate the MLF for every CPT as follows:

$$\mathrm{MLF}_{X_1} = \lambda_{1,1}\theta_{1,1,1} + \lambda_{1,2}\theta_{1,1,2}. \tag{4.6}$$

$$\mathrm{MLF}_{X_2} = \lambda_{1,1}\lambda_{2,1}\theta_{2,1,1} + \lambda_{1,1}\lambda_{2,2}\theta_{2,1,2} + \lambda_{1,2}\lambda_{2,1}\theta_{2,2,1} + \lambda_{1,2}\lambda_{2,2}\theta_{2,2,2}. \tag{4.7}$$

$$\mathrm{MLF}_{X_3} = \lambda_{1,1}\lambda_{3,1}\theta_{3,1,1} + \lambda_{1,1}\lambda_{3,2}\theta_{3,1,2} + \lambda_{1,2}\lambda_{3,1}\theta_{3,2,1} + \lambda_{1,2}\lambda_{3,2}\theta_{3,2,2}. \tag{4.8}$$

$$\mathrm{MLF}_{X_4} = \lambda_{2,1}\lambda_{3,1}\lambda_{4,1}\theta_{4,1,1} + \lambda_{2,1}\lambda_{3,1}\lambda_{4,2}\theta_{4,2,1} + ... + \lambda_{2,2}\lambda_{3,2}\lambda_{4,2}\theta_{4,4,2}. \tag{4.9}$$

Then we have $\mathrm{MLF}^1_{S_{12}}$ and $\mathrm{MLF}^2_{S_{12}}$ for C_1 and C_2 respectively:

$$\mathrm{MLF}^1_{S_{12}} = \lambda_{2,1}\lambda_{3,1}\beta^1_{x_{21},x_{31}} + \lambda_{2,1}\lambda_{3,2}\beta^1_{x_{21},x_{32}} + \lambda_{2,2}\lambda_{3,1}\beta^1_{x_{22},x_{31}} + \lambda_{2,2}\lambda_{3,2}\beta^1_{x_{22},x_{32}}. \tag{4.10}$$

$$\mathrm{MLF}^2_{S_{12}} = \lambda_{2,1}\lambda_{3,1}\beta^2_{x_{21},x_{31}} + \lambda_{2,1}\lambda_{3,2}\beta^2_{x_{21},x_{32}} + \lambda_{2,2}\lambda_{3,1}\beta^2_{x_{22},x_{31}} + \lambda_{2,2}\lambda_{3,2}\beta^2_{x_{22},x_{32}}. \tag{4.11}$$

By multiplying these MLFs, we get MLFs for Φ_{C_1} and Φ_{C_2} respectively as:

$$\begin{aligned}
\mathrm{MLF}_{C_1} &= \mathrm{MLF}_{X_1}\mathrm{MLF}_{X_2}\mathrm{MLF}_{X_3}\mathrm{MLF}^1_{S_{12}} && (4.12)\\
&= \lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1}\beta^1_{x_{21},x_{31}} + \lambda_{1,1}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2}\beta^1_{x_{21},x_{32}} && (4.13)\\
&+ \lambda_{1,1}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,2,1}\beta^1_{x_{22},x_{31}} + \lambda_{1,1}\lambda_{2,2}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,2,2}\beta^1_{x_{22},x_{32}} && (4.14)\\
&... && (4.15)\\
&+ \lambda_{1,2}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,4,1}\beta^1_{x_{22},x_{31}} + \lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,4,2}\beta^1_{x_{22},x_{32}}. && (4.16)\\
& && (4.17)\\
\mathrm{MLF}_{C_2} &= \mathrm{MLF}_{X_4}\mathrm{MLF}^2_{S_{21}} && (4.18)\\
&= \lambda_{2,1}\lambda_{3,1}\lambda_{4,1}\theta_{4,1,1}\beta^2_{x_{21},x_{31}} + \lambda_{2,1}\lambda_{3,1}\lambda_{4,2}\theta_{4,1,2}\beta^2_{x_{21},x_{31}} && (4.19)\\
&... && (4.20)\\
&+ \lambda_{2,2}\lambda_{3,2}\lambda_{4,1}\theta_{4,4,1}\beta^2_{x_{22},x_{32}} + \lambda_{2,2}\lambda_{3,2}\lambda_{4,2}\theta_{4,4,2}\beta^2_{x_{22},x_{32}}. && (4.21)
\end{aligned}$$

4.4.2 Message Passing with MLFs

If we can manipulate MLFs to compute the same components as in the message passing algorithm, we can accelerate the algorithm by compiling MLFs into ZDDs and utilizing ZDD techniques.

For the three types of variables $\lambda_{i,k}$, $\theta_{i,j,k}$, and $\beta^a_{S_{ab,\ell}}$ in MLFs as defined

above, $\lambda_{i,k} \in \{0,1\}$, $\theta_{i,j,k}, \beta^a_{s_{ab,\ell}} \in [0,1]$. During our method, parameter variables $\theta_{i,j,k}$ keep the values consistent to $P(x_{i,k} \mid \pi_{i,j})$ all the time. Values in message variables $\beta^a_{s_{ab,\ell}}$ dynamically change according to the messages passed between nodes $a, b \in N$. For any set of variables $W \subseteq V$, we define MLF_W representing probability distribution over W in the following sense. For any instantiation w of W, we can evaluate MLF_W so it returns the probability over w denoted by $\alpha_W(w)$.

Definition 3 The value of MLF_W at instantiation w, denoted by $\alpha_W(w)$, is the result of replacing each indicator variable $\lambda_{i,k}$ in MLF_W with 1 if $\lambda_{i,k}$ is consistent with w, and with 0 otherwise.

While using MLFs, messages over variables S_{ab} passed from a to b are obtained by calculating $\alpha_{C_a}(s_{ab,\ell})$ for all ℓ. The message passing algorithm with MLFs is summed up in Algorithm 4. It works in accordance with the message passing algorithm introduced in Section 2 except operations on potentials are transformed to operations on MLFs. We evaluate and change the values in message variables of MLFs to implement the same computation on potentials.

In the initialization step (line 2 in Algorithm 4), we initialize all message variables to 1. With this assignment, all MLFs satisfy the following equations which are consistent with the initialization in Algorithm 2 (line1-7):

$$\mathrm{MLF}_{C_a}(\beta^a_{s_{ab,\ell}} = 1) = \prod_{X_i \in SN(C_a)} P(X_i \mid \Pi_i), \text{ for all } a \in N, \tag{4.22}$$

$$\mathrm{MLF}^a_{S_{ab}}(\beta^a_{s_{ab,\ell}} = 1) = \sum_l \Lambda_{ab,\ell},\ \mathrm{MLF}^b_{S_{ab}}(\beta^b_{s_{ab,\ell}} = 1) = \sum_l \Lambda_{ab,\ell}, \text{ for all } (a,b) \in A. \tag{4.23}$$

Algorithm 4 MessagePassingwithMLF(T)

1: // Initialize
2: $\beta^{a}_{s_{ab,\ell}} \leftarrow 1, \beta^{b}_{s_{ab,\ell}} \leftarrow 1$, For all $(a,b) \in A$, and ℓ
3: // Collect messages : child to parent
4: **for** $d = d(T) - 1$ to 0 **do**
5: **for all** $b \in N_d$ **do**
6: **for all** $a \in child(b)$ **do**
7: Collect$(a, b, \text{MLF}_{C_a}, \text{MLF}_{C_b})$
8: **end for**
9: **end for**
10: **end for**
11: // Distribute messages : parent to child
12: **for** $d = 1$ to $d(T)$ **do**
13: **for all** $b \in N_d$ **do**
14: **for all** $a \in par(b)$ **do**
15: Distribute$(a, b, \text{MLF}_{C_a}, \text{MLF}_{C_b})$
16: **end for**
17: **end for**
18: **end for**
19: **return** evaluate all MLFs and return the results

Compared to the conventional algorithm, a message passing from node a to b in the collecting and distributing operations is performed differently. **In the message collecting operation, a message passing from a to b** done as follows:

- evaluate MLF_{C_a} on all instantiations of S_{ab} to get the messages $\alpha_{C_a}(s_{ab,\ell})$ for all ℓ, and preserve them in corresponding message variables $\beta^{a}_{s_{ab,\ell}}$ in MLF_{C_a} (refer to line 1 in Algorithm 5)

- Node b absorbs messages $\alpha_{C_a}(s_{ab,\ell})$ by updating corresponding messages variables $\beta^{b}_{s_{ab,\ell}}$ for all ℓ in MLF_{C_b}. (refer to line 3 in Algorithm 5)

Algorithm 5 Collect($a, b, \mathrm{MLF}_{C_a}, \mathrm{MLF}_{C_b}$)

1: $\beta^a_{s_{ab,\ell}} \leftarrow \alpha_{C_a}(s_{ab,\ell})$, for all ℓ // equivalent to $\phi_{s_{ab,\ell}} \leftarrow \sum_{c_{a,m} \setminus s_{ab,\ell}} \phi_{c_{a,m}}$ for all ℓ

2: $\beta^b_{s_{ab,\ell}} \leftarrow \frac{\alpha_{C_a}(s_{ab,\ell})}{\beta^b_{s_{ab,\ell}}}$, for all ℓ

In the message distributing operation, a message passing from a to b is evaluated as follows:

- evaluate MLF_{C_a} on all instantiations of S_{ab} to get messages $\alpha_{C_a}(s_{ab,\ell})$, and node b absorbs messages by updating message variables (refer to line 1 in Algorithm 6).

Algorithm 6 Distribute($a, b, \mathrm{MLF}_{C_a}, \mathrm{MLF}_{C_b}$)

1: $\beta^b_{s_{ab,\ell}} \leftarrow \frac{\alpha_{C_a}(s_{ab,\ell})}{\beta^b_{s_{ab,\ell}}}$, for all $(a, b) \in A$ and ℓ

Note that in the distributing operation, since a has absorbed messages in the collecting operation, there is no need to change the message variables in MLF_{C_a}, thus we only need to pass messages out to MLF_{C_b}.

After a full round of message passing, the MLFs for any Φ_{C_a} represents the joint distributions over variables in C_a:

$$\mathrm{MLF}_{C_a} = P(C_a)\,. \tag{4.24}$$

Also, if messages over S_{ab} are passed outward from a and collected inward to a, then we have:

$$\mathrm{MLF}^a_{S_{ab}} = P(S_{ab})\,. \tag{4.25}$$

Table 4.1 BN specifications for Chapter 4

Dataset	BN vertexes	Indicators	Parameters
ALARM	37	105	509
ASIA	8	16	18
HAILFINDER	56	223	2656
INSURANCE	27	89	984
HEPAR2	70	162	1453
WIN95PTS	76	152	574
PIGS	441	1323	5618
WATER	32	116	10083

Similarly, we can calculate the probability of any variable $X_i \in V$ by evaluating the MLF_{C_a} where $X_i \in C_a$. For the example in Fig 4.1, after performing the message passing algorithm on MLFs generated according to the procedure explained above, we can get the MLFs satisfying:

$$\mathrm{MLF}_{C_1} = P(X_1, X_2, X_3)\,, \mathrm{MLF}_{C_2} = P(X_2, X_3, X_4)\,, \tag{4.26}$$

$$\mathrm{MLF}^1_{S_{12}} = P(X_2, X_3)\,. \tag{4.27}$$

4.5 Experiment and Results

We implement our method on an Intel Core Quad CPU Q9550@2.83GHz * 4 PC with Ubuntu 12.04LTS and 3.8GiB of main memory. We manipulate up to 40,000,000 nodes of ZDDs using the ZDD package implemented by Minato [41]. We used dataset of BN Benchmark [23] ALARM, HAILFINDER, INSURANCE, etc. In our experiment. BN specifications with number of vertexes and parameters are shown in Table 4.1. The experiment results of our method comparing with conventional message passing algorithm are shown in Table 4.2 and Table 4.3.

We first show the time of generating a junction tree and time of message

Table 4.2 Exact inference with the junction tree algorithm.

Dataset	Junction tree algorithm				
	Clus -ters	Separ -ators	Compile (ms)	Maximum separator	Inference (ms)
ALARM	27	26	117	3(36)	56
ASIA	6	5	67	2(4)	5
HAILFINDER	43	42	160	4(297)	321
INSURANCE	19	18	114	6(2400)	4273
HEPAR2	58	57	123	6(96)	109
WIN95PTS	50	49	190	7(128)	103
PIGS	368	367	697	10(59049)	3148153
WATER	19	18	197	9(110592)	6119

Table 4.3 Exact inference with fast message passing algorithm.

Dataset	ZDD-based message passing			
	ZDD size	Message variables	Compile (ms)	Infer -ence(ms)
ALARM	6770	482	100	4
ASIA	278	64	4	1
HAILFINDER	45815	3116	3571	83
INSURANCE	359963	13712	6313	545
HEPAR2	18678	1376	336	67
WIN95PTS	26477	1756	548	31
PIGS	–	248922	–	–
WATER	–	515856	–	–

passing algorithm. We generated a junction tree for a BN using the heuristic algorithm known as *min-fill-in* [36]. We show the number of clusters and separators in a junction tree in the first and second columns. We show the maximum size of a separator in the third column. The first number in the third column is the number of variables in the maximum separator and the second number in the bracket is the number of all instantiations of these variables. The fourth column shows time for performing message passing algorithm once on the junction tree to get joint distribution over variables in clusters and separators.

The last four columns are the results of our proposed method with the size of ZDDs, time for generating ZDDs and time of message passing on ZDDs.

As the same in junction trees, we perform a full round message passing on ZDDs and get all MLFs representing joint distributions over variables in clusters and separators. And then we evaluate these MLFs to get these joint probabilities. We also show the number of message variables introduced in our approach which play an important role in our ZDD-based message passing algorithm. While passing messages with ZDDs, we need to evaluate all instantiations of separators. Thus time consumption of our method is linear time to the ZDD size and exponential to the size of separators. From the results, we can see that the inference with our ZDD-based message passing algorithm are more efficient comparing to the conventional one. Especially for the network INSURANCE, time for message passing accelerates about 8 times.

Unfortunately, for the networks PIGS and WATER, we could not conduct our method since the numbers of massage variables become too large for our ZDD package. This is mainly because there are too many message variables to generate ZDDs. While constructing a junction tree for a given BN, most research considers to construct a junction tree that has a treewidth as small as possible. However in our approach, the size of separators factors the most since we have to generate two message variables for every instantiation of the separators (See equation (4.4)). According to the result of message variable numbers, for the BN of WATER, it has only about 10000 parameters but we need to generate about 1 million message variables.

4.6 Summary

We proposed an improved method for exact inference in BNs to perform the message passing algorithm on ZDDs to accelerate the inference efficiency by exploiting local structures in the message passing algorithms. In

some cases, the junction tree with small clusters works well in the conventional message passing algorithm, but it gives rise to a considerable number of ZDDs in our method. This is because directly comping a junction tree into ZDDs, node sharing and cache memory can not be fully utilized because there are too many independent clusters sharing few local structures. What is more, decomposing a BN into too many clusters gives rise to the blow up of message variables. In the next chapter, we will introduce our proposed method to find proper d-separation structure to partition a given BNs into several conditional independent subgraphs and then compile these small graphs into ZDDs to further improve the exact inference.

Chapter 5

Separate Compilation of Bayesian Networks for Efficient Exact Inference

In this chapter, we introduce a method which combines the ideas in the previous chapters together to further improve the exact inference using ZDDs. We propose an approach of partitioning and separately compiling BNs. For every given BN, serial pattern *d*-separation sets are found and used to partition the BN into conditionally independent components. Separately compiling these components into ZDDs is more efficient than generating a giant ZDD for a whole network. However, partitioning a BN into too many components may give rise to considerable time consumption just as discussed in the previous chapter. Such time consumption grows exponentially with the number of vertexes used to partition

a BN. To trade off the off-line time consumption (for finding d-separations and compiling ZDDs) and on-line time consumption (for inference using ZDDs), the d-separations used to partitioning BNs are restricted to one-vertex and found using Tarjan's vertex-cut algorithm which can be performed linear time in the number of BN vertexes. The experiments illustrate that one-vertex d-separations exist in most BNs. Partitioning BNs with one-vertex d-separations improves the speed for both compilation and inference largely than the conventional ZDD-based method. To show the validity of partitioning with one-vertex d-separations, we also conduct the experiments of partitioning with two-vertex d-separations and the comparative experiments of junction tree algorithms.

5.1 Introduction

We have introduced two methods to improve the conventional ZDD-based method in the previous chapters. However, there are still some problems need to be taken into account in these two methods. First, for the factorization method in Chapter 3, though it can condense the ZDD size through factorizing a large ZDD into several small ones, their method is based on a large ZDD representing the whole BN using the conventional ZDD-based method. The original ZDD for the whole BN is too large and too time consuming to generate. Also, the d-separations used are found manually in an ad-hoc way. Moreover, they only compare the ZDD size with the conventional ZDD-based method but did not conduct experiments for exact inference, as they assume that computation

time of inference is linear to the ZDD size. Secondly, for the method in Chapter 4, they improve the conventional junction tree algorithm by compiling a junction tree into ZDDs and perform message passing on ZDDs. Through local structure exploiting and cache memory techniques, message passing algorithms for exact inference are indeed accelerated. But the main problem that junction tree with large clusters failed to be compiled into ZDDs still exists and exhibits the application of ZDD-based message passing algorithm.

In this chapter, we introduce a method which combines the two ideas together to solve these problems we met. We propose an idea of first partitioning a given BN into several conditionally independent components using serial pattern d-separations. Then, these components are separately compiled into ZDDs. Though there may be many serial pattern d-separations in a given BN, partitioning a BN into too many components would slow down the inference. To trade off the time between compilation and inference, the serial pattern d-separations are restricted to one-vertex and found using Tarjan's vertex-cut algorithm[52] which costs linear time in the number of BN vertexes. Comparing with the conventional ZDD-based method, the total ZDD size and time for compilation are largely reduced. The efficiency of inference using ZDDs is guaranteed by restricting the d-separation size to one vertex. To show the validity of partitioning with one-vertex d-separations, we also conduct the experiments of partitioning with two-vertex d-separations. The results show that one-vertex d-separations exist in most networks and partitioning with them works better than partitioning with two-vertex d-separations while performing on-line inference.

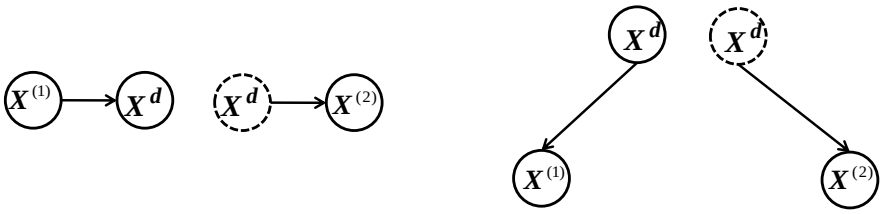
5.2 Partitioning BNs using d-Separations

5.2.1 Partitioning BNs

To resolve the problem of taking too long to constructing ZDDs for a large BN, naturally we think of partitioning the BN into a set of small components.

First, we would like to review the conception of d-separations in BNs. A *d-separation*[20] is a vertex set which is usually used to check conditional independence among vertexes in BN structure learning[25]. Let X_1, X_2 and X^d be three vertexes. Then their dependence is categorized in three graph patterns shown in Fig 3.1, the *serial pattern*, *diverging pattern* and *converging pattern*. If the three vertexes satisfy serial pattern d-separation and diverging pattern d-separation as shown in Fig 3.1, we say X^d is a d-separation of X_1 and X_2. This implies that if d-separation X^d is given, X_1 and X_2 become independent to each other (The converging pattern is normally treated as a d-connection: X_1 and X_2 becomes dependent if X^d is given).

The independence property prompts us the idea of partitioning a BN using d-separation vertex sets,that is, finding a vertex set $\boldsymbol{X}^d$ that partitions the BN vertexes into two disjoint vertex sets $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$ separately. We use $\boldsymbol{X}^{(m)}$ to indicate different vertex set with different $m \in \mathbb{N}^+$. If $\boldsymbol{X}^{(1)}, \boldsymbol{X}^{(2)}$ and $\boldsymbol{X}^d$ satisfy the d-separation patterns of (a) and (b), vertexes in $\boldsymbol{X}^{(1)}$ is independent to vertexes in $\boldsymbol{X}^{(2)}$ given $\boldsymbol{X}^d$. Then, the BN can be partitioned into two components as shown in 5.1.



(a) serial pattern (b) diverging pattern

Figure 5.1 Partitioning with d-separations.

Definition 4 *Partition a BN*: Given a BN $G = \langle V, A \rangle$, find a vertex set $\boldsymbol{X}^d$ to partition it into two components $G^1 = \left\langle \boldsymbol{X}^{(1)} \cup \boldsymbol{X}^d, A^1 \right\rangle$, $G^2 = \left\langle \boldsymbol{X}^{(2)} \cup \boldsymbol{X}^d, A^2 \right\rangle$ such that:

- $\boldsymbol{X}^{(1)} \cap \boldsymbol{X}^{(2)} = \emptyset$, $\boldsymbol{X}^{(1)} \cup \boldsymbol{X}^{(2)} \cup \boldsymbol{X}^d = V$.
- $\boldsymbol{X}^d$ d-separates $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$.
- $\forall X_i \in \boldsymbol{X}^{(1)}, \forall X_j \in \boldsymbol{X}^{(2)}$, X_i is a non-descendant of X_j.
- $A^1 = \{\langle X_i, X_j \rangle \mid \langle X_i, X_j \rangle \in A, X_i, X_j \in V^1\}$,
  $A^2 = \{\langle X_i, X_j \rangle \mid \langle X_i, X_j \rangle \in A, X_i, X_j \in V^2\}$,
  where $V^1 = \boldsymbol{X}^{(1)} \cup \boldsymbol{X}^d, V^2 = \boldsymbol{X}^{(2)} \cup \boldsymbol{X}^d$.

First, we consider the partitioning with serial pattern d-separation. After the partitioning shown in 5.1(a), the joint distribution of this BN is factorized as:

$$\begin{aligned} P\left(\boldsymbol{X}^{(1)},\boldsymbol{X}^d,\boldsymbol{X}^{(2)}\right) &= P\left(\boldsymbol{X}^{(2)} \mid \boldsymbol{X}^d\right) P\left(\boldsymbol{X}^d \mid \boldsymbol{X}^{(1)}\right) P\left(\boldsymbol{X}^{(1)}\right) \\ &= P\left(\boldsymbol{X}^{(2)} \mid \boldsymbol{X}^d\right) P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d\right). \end{aligned} \tag{5.1}$$

Our method separately compiles the MLFs for $P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d\right)$ and $P\left(\boldsymbol{X}^{(2)} \mid \boldsymbol{X}^d\right)$

into ZDDs. Marginal probability such as $P\left(\boldsymbol{X}^{(2)} = \boldsymbol{x}_l^{(2)}\right)$ can be computed by summing out irrelevant vertex set $\boldsymbol{X}^{(1)}$ in advance:

$$P\left(\boldsymbol{x}_l^{(2)}\right) = \sum_{\boldsymbol{X}^d} P\left(\boldsymbol{X}^{(2)} = \boldsymbol{x}_l^{(2)} \mid \boldsymbol{X}^d\right) \sum_{\boldsymbol{X}^{(1)}} P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d\right). \tag{5.2}$$

In the conventional ZDD-based method, it compiles $P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d, \boldsymbol{X}^{(2)}\right)$ into one large ZDD (according to equation (2.15)) and computes:

$$P\left(\boldsymbol{x}_l^{(2)}\right) = \sum_{\boldsymbol{X}^{(1)} \cup \boldsymbol{X}^d} P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d, \boldsymbol{X}^{(2)} = \boldsymbol{x}_l^{(2)}\right). \tag{5.3}$$

Comparing with the conventional ZDD-based method, first, our method reduces time consumption of compiling $P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d, \boldsymbol{X}^{(2)}\right)$ by separately compiling $P\left(\boldsymbol{X}^{(2)} \mid \boldsymbol{X}^d\right)$ and $P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^d\right)$. Secondly, the computation for $P\left(\boldsymbol{x}_l^{(2)}\right)$ are accelerated by summing out irrelevant vertexes $\boldsymbol{X}^{(1)}$ in advance. Thirdly, if the on-line inference is only about vertexes $\boldsymbol{X'}^{(2)} \subset \boldsymbol{X}^{(2)}$ which is computed using the probability $P\left(\boldsymbol{X}^d\right)$, it can be accelerated through this partitioning by first computing and storing $P\left(\boldsymbol{X}^d\right)$ off-line.

Next, we consider the partitioning with diverging pattern d-separation. Partitioning in 5.1(b) means that if we partition this BN with vertex set $\boldsymbol{X}^d$, its descendant vertexes in different branch with $\boldsymbol{X}^d$ form different components. While compiling them into ZDDs, we construct ZDDs for $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$ separately only based on the ZDDs of their common ancestor $\boldsymbol{X}^d$. Note the ZDD construction strategy is the same to the conventional ZDD-based method. Therefore, in our method, we only consider to use the serial pattern d-separation to partition a BN.

5.2.2 On-line Inference Time Consumption

For any given BN, it can be recursively partitioned into several components to further reduce ZDD size until no serial pattern d-separations are found. One main problem is that as the number of components increases, time for on-line inference may increase unacceptably.

As the partitioning shown in equation (5.2), to perform the summing over operations$\sum_{\boldsymbol{X}^{(1)}} P\left(\boldsymbol{X}^{(1)}, \boldsymbol{X}^{d}\right)$, marginal probabilities $P\left(\boldsymbol{X}^{d}\right)$ need to be calculated first. After compiling them into ZDDs, this calculation can be implemented by evaluating all the ZDD paths containing variables corresponding to $P\left(\boldsymbol{x}_{l}^{d}\right)$ for all $\boldsymbol{x}_{l}^{d} \in \mathrm{Dom}(\boldsymbol{X}^{d})$. These evaluations may lead to considerable time consumption which is exponential to $|\mathrm{Dom}(\boldsymbol{X}^{d})|$, the number of instantiations over vertexes in $\mathrm{Dom}(\boldsymbol{X}^{d})$. Thus partitioning a BN with a d-separation consisting of too many vertexes will lead to a blow up in the number of instantiations in the d-separation. As a result, the best choice of a serial pattern d-separation would be the vertex sets that can result in a partitioning with minimum $|\mathrm{Dom}(\boldsymbol{X}^{d})|$ and minimum ZDD size. Find a set of such d-separations is a very complex task.

To trade-off time consumption for compiling and inference, we decided to use only the set of all d-separations consisting of one vertex because they are highly effective and easily extracted without any complicated evaluation. We use *one-vertex d-separation* to denote the d-separation consisting of only one vertex. To enumerate all one-vertex d-separations, we use Tarjan's vertex-cut algorithm which can be carried out in linear time in the number of BN vertexes. Using all one-vertex d-separations, a BN in partitioned into several condition-

ally independent components. The experiments will show that most of BNs contain one-vertex d-separations. Partitioning BNs using all the one-vertex d-separations can improve both compilation and inference a lot and even works better than partitioning with *two-vertex d-separations*, d-separations consisting of two vertexes.

5.2.3 Enumerating One-vertex d-Separations

Tarjan's vertex-cut algorithm [52] is known as one of the most famous algorithms used in an undirected graph to enumerate all cut vertexes (a vertex such that deleting edges connected to the vertex can decompose an undirected graph into two or more components). The algorithm is implemented in time linear in the number of vertexes in the given graph based on the idea of Depth First Search (DFS). In our method, we use this algorithm on an equivalent undirected graph of the given BN to get a set of cut vertexes that contains all one-vertex d-separation candidates. Then we pick out the serial pattern d-separations and use all of them to partition a BN.

To use Tarjan's vertex-cut algorithm, first we need to transfer a BN into its equivalent undirected form which is known as *Moral Graph*.

Definition 5 The Moral Graph $M(V)$ of a Bayesian Network G is the undirected graph over V that contains an undirected edge between vertexes X_i and X_j if:

- there is a directed edge between them (in either direction) or
- X_i and X_j are both parents of the same vertex [31].

Given a BN, first we add edges between very pair vertexes sharing the same children. Then we delete directions for all edges. A BN is therefore moralized into a corresponding moral graph $M(V)$. We use $M(V)$ as the input of Tarjan's vertex-cut algorithm and run this algorithm once on $M(V)$. We can get the output of a vertex set $\boldsymbol{X^s}$. Thus, $\forall X_i \in \boldsymbol{X^s}$ is a cut vertex of $M(V)$ such that deleting edges connected to X_i separates $M(V)$ into several components. Details for the algorithms are summed up in Algorithm 7 and 8. Having all the cut vertexes, next we explain that vertexes in different components of $M(V)$ separated by these cut vertexes are independent to each other.

Algorithm 7 Tarjan's vertex-cut algorithm

Input:
$M(V)$;
Output:
set of cut vertexes $\boldsymbol{X}^s$;

1: $\boldsymbol{X}^s = \emptyset$;
2: **for** $X_i \in V$ **do**
3: $parent[X_i]$=NULL; $cut[X_i]$=FALSE;
4: $visit[X_i]$=-1;
5: **end for**
6: $time$=0;
7: **for** $X_i \in V$ **do**
8: **if** $visit[X_i]$==-1; **then**
9: DFS_Visit(X_i);
10: **end if**
11: **end for**
12: **for** $X_i \in V$ **do**
13: **if** $cut[X_i]$ ==TRUE **then**
14: $\boldsymbol{X}^s = \boldsymbol{X}^s \cup X_i$;
15: **end if**
16: **end for**
17: **return** $\boldsymbol{X}^s$

Algorithm 8 DFS_Visit(X_i)

1: $low[X_i] = d[X_i] =$++$time$
2: $visit[X_i] = 0$, $children[X_i] = 0$
3: **for** $X_i' \in Adj(X_i)$ **do**
4: **if** $visit[X_i'] ==$-1 **then**
5: $children[X_i]$++, $parent[X_i'] = X_i$
6: DFS_Visit(X_i')
7: $low[X_i] = min(low[X_i'], low[X_i])$
8: **if** $parent[X_i]!$ =NULL and $low[X_i'] \geqslant d[X_i]$ **then**
9: $cut[X_i]$ =TRUE
10: **end if**
11: **if** $parent[X_i] ==$NULL and $children[X_i] > 1$ **then**
12: $cut[X_i]$ =TRUE
13: **end if**
14: **else**
15: **if** $visit[X_i'] == 0$ and $parent[X_i] \neq X_i'$ **then**
16: $low[X_i] = min(low[X_i], d[X_i'])$
17: **end if**
18: **end if**
19: **end for**
20: $visit[X_i] = 1$

Theorem 1 Let $\boldsymbol{X}^{(1)}$, $\boldsymbol{X}^{(2)}$,$\boldsymbol{X}^{(3)}$ be three disjoint vertex sets in a BN and $V = \boldsymbol{X}^{(1)} \cup \boldsymbol{X}^{(2)} \cup \boldsymbol{X}^{(3)}$. We say that $\boldsymbol{X}^{(1)}$ is independent to $\boldsymbol{X}^{(2)}$ given $\boldsymbol{X}^{(3)}$ if $\boldsymbol{X}^{(1)}$ is separated from $\boldsymbol{X}^{(2)}$ by $\boldsymbol{X}^{(3)}$ in $M(V)$ [31].

According to the Theorem 1, vertexes in different components decomposed by $X_i \in \boldsymbol{X}^s$ are independent to each other. Now we know that every $X_i \in \boldsymbol{X}^s$, $\{X_i\}$ can be treated as a separation which decomposes the BN into several conditionally independent components. Next we check whether $\{X_i\}$ and these components satisfies serial pattern or not.

For every decomposition by $\{X_i\}$, vertex sets $\boldsymbol{X}^{(1)}$ (where $\boldsymbol{X}^{(1)} \cap \Pi_i \neq \emptyset$) are treated as the upstream component, and all other vertexes are treated as

the downstream component $\boldsymbol{X}^{(2)}$. Then, $\{X_i\}$ is a serial pattern d-separation $\boldsymbol{X}^d$ such that:

- There are no streams flowing up to the upstream components from X_i: $\nexists X_j \in \boldsymbol{X}^{(1)}, X_i \in \Pi_j$ or
- There are no streams flowing up to X_i from downstream components: $\nexists X_j \in \boldsymbol{X}^{(2)}, X_j \in \Pi_i$.

Using above checking rules, we can pick out all the one-vertex serial pattern d-separations from the separation vertex set $\boldsymbol{X}^s$. Then, we use them all to recursively partition a given BN into several components. For the example in 5.2(a), its corresponding moral graph is shown in (b). Vertexes X_3 and X_5 are found as cut vertexes using Tarjan's vertex-cut algorithm. Through the checking rules, they are both confirmed as serial pattern d-separations. Then we use this two vertexes to recursively partition a BN into three components as shown in 5.2(c). Note that no matter in what order we use these one-vertex d-separations, we can always get the same partitioning. If we iteratively partition this BN by choosing X_3 first and then choosing X_5, we get the same partitioning as using X_5 first. However, for vertex sets $\{X_3, X_4\}$ and $\{X_4, X_5\}$ in 5.2 which are both two-vertex serial pattern d-separations, if we consider two-vertex d-separations, we have to choose one using some complicated evaluation since partitioning with either of them leads to different results. This is another advantage of restricting serial pattern d-separations to one-vertex.

5.3 Separate Compiling and On-line Inference

In this section, we introduce how to compile these components into MLFs so that we can construct ZDDs and introduce how to use ZDDs to execute exact inference.

5.3.1 MLFs for Independent Components

While partitioning a BN into G^1, G^2 with d-separation $\boldsymbol{X}^d$ as shown in 5.1(a), we take the same idea with the conventional ZDD-based method that constructs a ZDD for one vertex depending on its parents' ZDDs. Thus, for the vertexes in G^1, MLFs for them are the same as defined in equations (7), (8) and (9). The difference is the MLFs for the vertexes in G^2. For the vertex who does not have $\boldsymbol{X}^d$ as its parent, the MLF for it is the same as defined in the conventional ZDD-based method. But for vertexes which have $\boldsymbol{X}^d$ as their parents, if $X_i \in V^2$ only has $\boldsymbol{X}^d$ as its parents, the MLF for X_i is:

$$\mathrm{MLF}_{X_i} = \sum_{k:x_{i,k}\in \mathrm{Dom}(\{X_i\})} \mathrm{MLF}_{x_{i,k}}. \tag{5.4}$$

$$\mathrm{MLF}_{x_{i,k}} = \sum_{j:x_j^d \in \mathrm{Dom}(\boldsymbol{X}^d)} \lambda_{i,k}\theta_{i,j,k}\mathrm{MLF}_{x_j^d}, \tag{5.5}$$

where $X_i \in V^2$ and $\Pi_i = \boldsymbol{X}^d$.

If $X_i \in V^2$ also has other vertexes as its parents, then its MLF is given

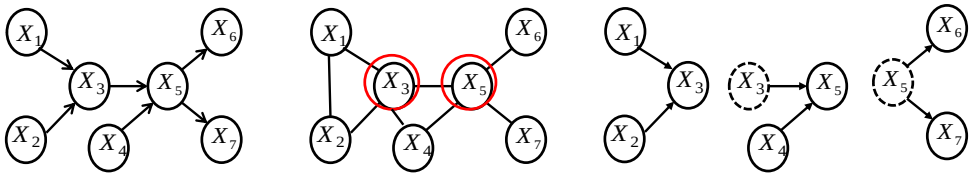


Figure 5.2 Example of partitioning.

by:

$$\mathrm{MLF}_{X_i} = \sum_{k:x_{i,k}\in \mathrm{Dom}(\{x_i\})} \mathrm{MLF}_{x_{i,k}} \tag{5.6}$$

$$\mathrm{MLF}_{x_{i,k}} = \sum_{\substack{j\pi_{i,j}\in \mathrm{Dom}(\Pi_i) \\ l\boldsymbol{x}_l^d \in \mathrm{Dom}(\boldsymbol{X}^d)}} \lambda_{i,k}\theta_{i,j,k}\mathrm{MLF}_{\boldsymbol{x}_l^d} \prod_{i',k':x_{i',k'}\in\pi_{i,j}\backslash\boldsymbol{x}_l^d} \mathrm{MLF}_{x_{i',k'}} \tag{5.7}$$

where $X_i \in V^2$, $\boldsymbol{X}^d \subset \Pi_i$ and $\boldsymbol{x}_l^d$ is consistent with $\pi_{i,j}$

For the MLF of $\boldsymbol{X}^d = \{X_i\} \subset V^2$, we introduce $\boldsymbol{\beta}_i \triangleq \{\beta_{i,l}\}_l$ as *separator variables* that map each instantiation of $\boldsymbol{X}^d$ into a real number. we define $\mathrm{MLF}_{\boldsymbol{X}^d}$ as:

$$\mathrm{MLF}_{\boldsymbol{X}^d} = \sum_l \mathrm{MLF}_{\boldsymbol{x}_l^d},\ \mathrm{MLF}_{\boldsymbol{x}_l^d} = \lambda_{i,l}\beta_{i,l}, \text{where } \boldsymbol{X}^d = \{X_i\}. \tag{5.8}$$

Note that $\beta_{i,l}$ dynamically changes and takes the values calculated from up-stream components according to different queries:

$$\beta_{i,l} \leftarrow \mathrm{MLF}_{x_{i,l}}, \text{where } \{X_i\} = \boldsymbol{X}^d \tag{5.9}$$

Given a BN, after partitioning it with a set of $\boldsymbol{X}^d$, we generate ZDDs $Z(X_i)$ for every $X_i \in V$ in the BN. In addition, we generate ZDDs $Z(\boldsymbol{X}^d)$ for every $\boldsymbol{X}^d$.

For the example in 5.2, suppose every vertex contains two values. For

the vertexes $\{X_1, X_2, X_3\}$ in the first component, we have MLFs in the same way as that of conventional ZDD-based method. Then for the vertexes in the second component, MLF_{X_4} is generated the same to conventional ZDD-based method. The MLF for d-separation vertex X_3 is:

$$\mathrm{MLF}_{x^d_{3,1}} = \beta_{3,1}\lambda_{3,1},\ \mathrm{MLF}_{x^d_{3,2}} = \beta_{3,2}\lambda_{3,2}. \tag{5.10}$$

Then,

$$\begin{aligned}\mathrm{MLF}_{X_5} =& \lambda_{5,1}(\theta_{5,1,1}\mathrm{MLF}_{x^d_{3,1}}\mathrm{MLF}_{x_{4,1}} + \theta_{5,2,1}\mathrm{MLF}_{x^d_{3,2}}\mathrm{MLF}_{x_{4,1}}) \\ &+ \lambda_{5,2}(\theta_{5,1,2}\mathrm{MLF}_{x^d_{3,1}}\mathrm{MLF}_{x_{4,2}} + \theta_{5,2,2}\mathrm{MLF}_{x^d_{3,2}}\mathrm{MLF}_{x_{4,2}}).\end{aligned} \tag{5.11}$$

Similarly, for the third component, MLFs for X_5, X_6 and X_7 are:

$$\mathrm{MLF}_{x^d_{5,1}} = \beta_{5,1}\lambda_{5,1}, \mathrm{MLF}_{x^d_{5,2}} = \beta_{5,2}\lambda_{5,2}. \tag{5.12}$$

$$\begin{aligned}\mathrm{MLF}_{X_6} =& \lambda_{6,1}(\theta_{6,1,1}\mathrm{MLF}_{x^d_{5,1}} + \theta_{6,2,1}\mathrm{MLF}_{x^d_{5,2}}) \\ &+ \lambda_{6,2}(\theta_{6,1,2}\mathrm{MLF}_{x^d_{5,1}} + \theta_{6,2,2}\mathrm{MLF}_{x^d_{5,2}}).\end{aligned} \tag{5.13}$$

$$\begin{aligned}\mathrm{MLF}_{X_7} =& \lambda_{7,1}(\theta_{7,1,1}\mathrm{MLF}_{x^d_{5,1}} + \theta_{7,2,1}\mathrm{MLF}_{x^d_{5,2}}) \\ &+ \lambda_{7,2}(\theta_{7,1,2}\mathrm{MLF}_{x^d_{5,1}} + \theta_{7,2,2}\mathrm{MLF}_{x^d_{5,2}}).\end{aligned} \tag{5.14}$$

5.3.2 Exact Inference with ZDDs

As we have showed in equation (2.8), for the exact inference in a BN, if we can efficiently calculate $P\left(\boldsymbol{x}_l^{(1)}, \boldsymbol{x}_k^{(2)}\right)$ and $P\left(\boldsymbol{x}_k^{(2)}\right)$, the probability of this query can be easily obtained. If the variables to be inferred and variables observed are in the same component, the probability can be calculated the same as the conventional ZDD-based method. The problem is when these variables

are in different components, we need to take efforts on separation variables. Whenever we visit a ZDD node representing a separate variable β, we get its value by tracing corresponding ZDDs to collect the information passed from other components.

Let's considering the following example of calculating $P(x_{2,1}, x_{5,1})$ with the BN in 5.2(c), . First, we have:

$$\begin{aligned} \mathrm{MLF}_{x_{5,1}} &= \lambda_{3,1}\lambda_{4,1}\lambda_{5,1}\beta_{3,1}\theta_{4,1,1}\theta_{5,1,1} \\ &\quad + \lambda_{3,2}\lambda_{4,1}\lambda_{5,1}\beta_{3,2}\theta_{4,1,1}\theta_{5,2,1} \\ &\quad \cdots \\ &\quad + \lambda_{3,2}\lambda_{4,2}\lambda_{5,1}\beta_{3,2}\theta_{4,1,2}\theta_{5,4,1}. \end{aligned} \tag{5.15}$$

By setting λs to 1, we get:

$$\begin{aligned} \mathrm{MLF}_{x_{5,1}} &= \beta_{3,1}\theta_{4,1,1}\theta_{5,1,1} \\ &\quad + \beta_{3,2}\theta_{4,1,1}\theta_{5,2,1} \\ &\quad \cdots \\ &\quad + \beta_{3,2}\theta_{4,1,2}\theta_{5,4,1}. \end{aligned} \tag{5.16}$$

For parameter variable θs, they take values as in CPT. For separator variable $\beta_{3,1}$ and $\beta_{3,2}$, we trace the upstream ZDD to get the information. Since we already have:

$$\begin{aligned} \mathrm{MLF}_{X_2} &= \mathrm{MLF}_{x_{2,1}} + \mathrm{MLF}_{x_{2,2}} \\ &= \lambda_{2,1}\theta_{2,1,1} + \lambda_{2,2}\theta_{2,1,2}, \end{aligned} \tag{5.17}$$

$$\begin{aligned} \mathrm{MLF}_{X_3} &= \mathrm{MLF}_{x_{3,1}} + \mathrm{MLF}_{x_{3,2}} \\ &= \lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} \\ &\quad + \lambda_{1,1}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2} \\ &\quad \cdots \\ &\quad + \lambda_{1,2}\lambda_{2,2}\lambda_{3,2}\theta_{1,1,2}\theta_{2,1,2}\theta_{3,4,2}. \end{aligned} \tag{5.18}$$

According to the query, we generate:

$$\begin{aligned} \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}} &= \lambda_{1,1}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} \\ &\quad + \lambda_{1,2}\lambda_{2,1}\lambda_{3,1}\theta_{1,1,2}\theta_{2,1,1}\theta_{3,2,1}, \end{aligned} \tag{5.19}$$

$$\begin{aligned} \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,2}} &= \lambda_{1,1}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2} \\ &\quad + \lambda_{1,2}\lambda_{2,1}\lambda_{3,2}\theta_{1,1,2}\theta_{2,1,1}\theta_{3,2,2}. \end{aligned} \tag{5.20}$$

By setting λs to 1, we can get:

$$\begin{aligned} \beta_{3,1} &\leftarrow \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,1}} \\ &= \theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,1} + \theta_{1,1,2}\theta_{2,1,1}\theta_{3,2,1}, \end{aligned} \tag{5.21}$$

$$\begin{aligned} \beta_{3,2} &\leftarrow \mathrm{MLF}_{x_{2,1}}\mathrm{MLF}_{x_{3,2}} \\ &= \theta_{1,1,1}\theta_{2,1,1}\theta_{3,1,2} + \theta_{1,1,2}\theta_{2,1,1}\theta_{3,2,2}. \end{aligned} \tag{5.22}$$

Substituting $\beta_{3,1}, \beta_{3,2}$ into $\mathrm{MLF}_{x_{5,1}}$, we can get $P(x_{2,1}, x_{5,1})$.

By using the same multiplication algorithm as in the conventional ZDD-based method, no term can contain the same variable more than once. Contradicting terms are automatically eliminated. We can always generate ZDDs containing variables only related to the query so that unnecessary calculations are avoided. An important point is that our idea of partitioning is independent to queries. Once we partition a BN with d-separations and generate ZDDs, we can use them to calculate probabilities for any queries.

One may question that whether this calculation is efficient enough for calculating the conditional probability such as $P(x_{2,1} \mid x_{5,1})$ since we have to calculate $P(x_{5,1})$ and $P(x_{2,1}, x_{5,1})$ first to get the final results. Note that

$$P(x_{2,1}, x_{5,1}) = \sum_{X_1,X_3,X_4} P(X_1, X_2 = x_{2,1}, X_3, X_4, X_5 = x_{5,1}), \tag{5.23}$$

$$\begin{aligned} P(x_{5,1}) &= \sum_{X_1,X_3,X_4} P(X_1, X_2 = x_{2,1}, X_3, X_4, X_5 = x_{5,1}) \\ &\quad + \sum_{X_1,X_3,X_4} P(X_1, X_2 = x_{2,2}, X_3, X_4, X_5 = x_{5,1}). \end{aligned} \tag{5.24}$$

Table 5.1 BN specifications for Chapter 5

Dataset	BN specifications		
	Vertexes	Indi-cators	Para-meters
ALARM	37	105	509
WIN95PTS	76	152	574
HEAPR2	70	162	1,453
HAILFINDER	56	223	2,656
PATHFINDER	135	520	2,304
INSURANCE	27	89	984
MILDEW	35	616	6,709
WATER	32	116	3,578
PIGS	441	592	5,618
BARLEY	48	84	114,005
DIABETES	413	4,682	17,622

After we calculate $P(x_{2,1}, x_{5,1})$, we only need to calculate the second part in formula (5.24) for $P(x_{5,1})$. Repetitive calculations can be avoided using the cache memory technique in ZDDs.

5.4 Experiments and Results

5.4.1 Overview

We conducted our experiments using the benchmark networks [23] and the ZDD package implemented by Minato [41] on an intel Core i7-2700k CPU @ 3.50ghz × 8PC with Ubuntu 16.04LTS and 31.4Gib of main memory. The network specifications such as BN name, the number of vertexes, indicators, and parameters are shown in Table 5.1.

The results of our proposed method comparing with the conventional ZDD-method is shown in Table 5.3. Table 5.2 is the results for the conventional ZDD-based method. For a given BN, first, we find one-vertex d-separations and

Table 5.2 Experiment results for the conventional ZDD-based method

Dataset	Conventional ZDD-based method [42]			
	ZDD Size	Compil-ing(ms	Mar. (ms)	Arb. (ms)
ALARM	34,299	57	9	651
WIN95PTS	26,477	104	15	231
HEPAR2	51,000	126	12	885
HAILFINER	294,608	467	39	3,330
PATHFINDER	31,549	2,633	14	483
MILDEW	15,310,511	664,097	17,545	522,670
WATER	25,629	874	12	175
PIGS	73,715	517	28	97
BARLEY	time out	time out	time out	time out
DIABETES	time out	time out	time out	time out
WIN95PTS (Problem1)	24,460	84	7	248
BARLEY (ngtilg)	1,091,779	4,360	354	461,975

Table 5.3 Experiment results for partitioning with all one-vertex d-separations

Dataset	Proposed method					
	ZDD Size	Compil-ing(ms)	Mar. (ms)	Arb. (ms)	Time (d-sep) (ms)	No. of d-sep
ALARM	10,291	26	2	106	15	5(27,5,3,2)
WIN95PTS	27,397	107	7	79	6	4(71,3,2)
HEPAR2	40,308	77	6	281	15	7(48,9,5,4,2)
HAILFINER	188,088	317	33	1,753	28	6(45,4,3,2)
PATHFINDER	31,553	2,621	20	461	3	1(108,2)
MILDEW	15,310,711	688,067	8,786	225,250	1	1(34,2)
WATER	25,629	874	12	175	1	0
PIGS	48,588	509	23	69	198	41 (324,13,8,6,4,3)
BARLEY	time out	time out	time out	time out	0.52	2(46,3)
DIABETES	time out	time out	time out	time out	23	2(409,3)
WIN95PTS (Problem1)	5,299	62	4	46	1	3(26,8,3,2)
BARLEY (ngtilg)	1,089,670	4,049	171	222,881	55	2(15,4,2)

Table 5.4 Results of partitioning with all d-separations consisting of one or two vertexes

Dataset	Partitioning using all possible one- and two-vertex d-separations			
	ZDD size	Compiling (ms)	Mar.(ms)	No.of d-sep
ALARM	1,304	10	5	11
WIN95PTS	6,812	55	5	10
HEPAR2	5,544	44	10	17
HAILFINER	29,756	84	90	10
PATHFINDER	31,553	2,621	20	1
MILDEW	7,440,861	740,035	98,081	3
WATER	25,670	821	13	3
PIGS	13,829	487	87	73
DIABETES	time out	time out	time out	6
BARLEY	time out	time out	time out	2

Table 5.5 Results of partitioning with heuristically chosen d-separations consisting of one or two vertexes

Dataset	Partitioning with the best greedy selection from one and two-vertex d-separations			
	ZDD size	Compiling (ms)	Mar.(ms)	No.of d-sep
ALARM	2,351	26	2	5
WIN95PTS	7,120	87	4	6
HEPAR2	9,020	82	3	10
HAILFINER	151,048	333	37	7
PATHFINDER	31,553	2,621	20	1
MILDEW	9,922,853	190,564	111,615	3
WATER	25,643	869	12	2
PIGS	68,441	462	30	22
DIABETES	time out	time out	time out	time out
BARLEY	time out	time out	time out	time out

Table 5.6 Results for jointree algorithm: min-fill

Dataset	Min-fill triangulation			
	Cluster	Separators (maximum)	Mar. (ms)	Compiling(ms)
ALARM	27	36	30	55
WIN95PTS	50	128	74	100
HEPAR2	58	96	79	87
HAILFINER	43	297	250	86
PATHFINDER	91	8,064	33,397	265
MILDEW	29	28,800	time out	359
WATER	19	110,592	time out	131
PIGS	368	59,049	time out	168
BARLEY	36	1,125,600	time out	362
DIABETES	335	14,025	time out	607

Table 5.7 Results for jointree algorithm:min-degree

Dataset	Min-degree triangulation			
	Cluster	Separators (maximum)	Mar. (ms)	Compiling(ms)
ALARM	27	48	39	66
WIN95PTS	50	128	88	78
HEPAR2	58	96	74	72
HAILFINER	43	297	244	87
PATHFINDER	91	4,800	13,257	270
MILDEW	29	87,840	time out	415
WATER	19	442,368	time out	118
PIGS	368	531,441	time out	184
BARLEY	36	907,200	time out	296
DIABETES	336	39,270	time out	620

use them all to partition the BN. Time for finding d-separations and partitioning are together shown in column "Time(d-sep)(ms)". The number of d-separations we use are shown in the last column before the bracket. In the off-line compilation, we generate ZDDs for this BN. Then, we calculate values of separator variables βs by setting all λs to 1 in advance. Cache memories for these calculations would help to improve on-line inference. The ZDD size is shown in

Table 5.8 Results for CNF based method

Dataset	CNF based method in [8]			
	AC Nodes	AC Edges	Compiling(ms)	Mar. (ms)
ALARM	1,570	2,848	5	2
WIN95PTS	3,004	5,638	9	3
HEPAR2	7,697	11,966	10	4
HAILFINER	8,594	16,532	11	4
PATHFINDER	17,825	33,786	35	9
MILDEW	1,118,179	2,219,244	241	53
WATER	28,741	56,496	54	12
PIGS	625,752	1,248,854	99	43
BARLEY	18,196,115	36,348,928	1,448	489,891
DIABETES	7,655,537	15,300,530	828	229

the first column. Time of generating ZDDs and calculating βs are shown as "Compiling(ms)" represented in the second column. For the on-line inference, we conduct two kinds of exact inference. First is to calculate the marginal probability of every vertex in the BN. Time for the this inference is shown in column "Mar.(ms)". The second is to infer one hundred of instantiations. we randomly selected hundred pairs of BN vertexes and randomly instantiate them. Then we calculate marginal probabilities over these instantiations. The total time of computation for these instantiations are presented in the column named "Arb.(ms)". Time limitation in our experiments is set within 30 minutes.

To show the validity of partitioning with one-vertex d-separations, we also conducted the experiments of partitioning a BN using one- or two-vertex d-separations in Table 5.4 and Table 5.5.

5.4.2 Results and Discussion

According to Table 5.3, most of the BNs contain one-vertex serial pattern d-separations. Our method performs quite well than the conventional ZDD-based method on condensing ZDD size for networks such as ALARM, HEPAR2 and PIGS, which present obviously layer-wised structures. Also, both time of compilation and inference are reduced significantly on these networks. But network WATER does not contain one-vertex serial pattern d-separations. Vertexes in this network are structured just in 4 layers and closely connect to each other. For networks such as MIDLEW and PATHFINDER, there are just few one-vertex serial pattern d-separations. Some of the d-separations are located in the corner of the networks so that partitioning with these d-separations does not bring much improvement while compiling them into ZDDs. Also, the inference time for hundred vertex pairs shown in column "Arb(ms)" are largely reduced through reusing the results of probabilities over d-separations based on the cache memory technique. An example of partitioning ALARM is shown in 5.3.

One may question the validity of partitioning only with one-vertex d-separations because intuitively, d-separations which can decompose a BN into two balanced components are considered to be more likely to generate small ZDDs. However, if the d-separation consists of too many vertexes, inference time after partitioning may be unacceptable because inference time greatly depends on $|\mathrm{Dom}(\boldsymbol{X}^d)|$ which grows exponentially with the number of vertexes in $\boldsymbol{X}^d$ (suppose every vertex is binary valued). By restricting d-separation to one-vertex, partitioning with all one-vertex d-separations is always highly effective in both ZDD size and inference efficiency. In the last column in Ta-

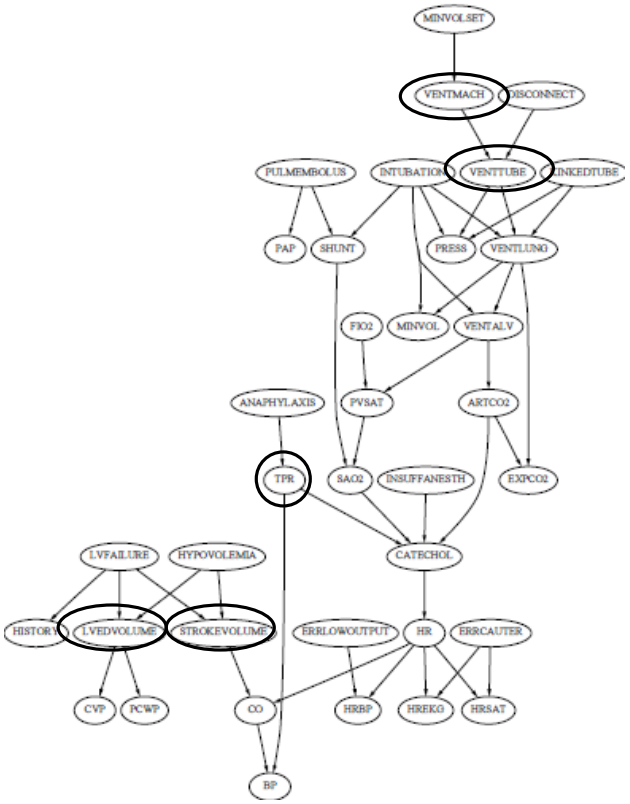


Figure 5.3 Example of ALARM

ble 5.3, we list the different numbers of component sizes partitioned by the one-vertex d-separations in the bracket. For the example of ALARM in Figure 8, it is partitioned into 6 components with size 27, 5, 3, 3, 2, 2 respectively, we add (27,5,3,2) in the last column in Table 2. Note for networks of ALARM, WIN95PTS, HEPAR2, HAILFINDER, and PIGS, although they are not partitioned into very balanced components, by cutting off some small components using one-vertex d-separations, ZDD size are still reduced a lot so that inference time is also improved. Also, for networks of MILDEW and PATHFINDER which are partitioned into apparently unbalanced large and small components, the efficiency of compilation and inference is not badly affected a lot because the increasing of ZDD nodes are just for compiling the separator variables

whose number are controlled by limiting the d-separations to one-vertex.

To present the validity of partitioning with all one-vertex d-separations, we also conducted the experiments of partitioning with two-vertex d-separations shown in Table 5.4. First all one- and two-vertex serial pattern d-separations are enumerated using a complete search approach which costs time about $O(|V|^3)$. Then a serial pattern d-separation is chosen as the suitable one such that partitioning with this d-separation could give rise to the largest reductions of the number of MLF items, which is an approximation of ZDD size. Using this d-separation, the given BN is partitioned into two components and separately compiled into ZDDs. We repeated this procedure for these components until no suitable d-separations are found. We generated ZDDs and performed the inference for every iteration to see how the ZDD size and inference time change as the number of components increases. Table 5.4 are results for iteratively partitioning BNs using all possible one- and two-vertex d-separations until no suitable d-separations are found. According to results, the ZDD size and time for compilation are largely reduced through partitioning BNs also with two-vertex d-separations, which is more likely to partition a BN into balanced components than one-vertex d-separations. However, inference time may increase incredibly such as HAILFIDER, MILDEW and PIGS.

What is more, we also present the results that we do not partitioning a BN to the end until no suitable d-separations are found, but stop the partitioning at the point that the inference is the fastest. These results are shown in the Table 5.5. Though we stop the partitioning at a suitable stage, ZDD size and inference time may be reduced for networks such as WIN95PTS and HEPAR2. But we have to generate ZDDs and perform the inference repeatedly to the end to choose the best result. Time consumption for such work is unacceptable.

Also partitioning on network such as MILDEW still performs rather badly. It is worthy of finding a good measurement of choosing suitable d-separation and precise trade-off between time for compiling and inference as the future work. At the present stage, our idea of using only one-vertex d-separations proposed is a simple and easy-to-use heuristic method. What is more, it has no risk of extremely reducing the inference efficiency.

One may query what if appropriate one-vertex serial pattern d-separations do not generally exist in the practical BNs. Note one main advantage of ZDD-based compiling approach is that when we query the probability of a vertex, we can only focus on the subgraph which consists of the concerned vertex and all its ancestors. Therefore, we do not have to find the d-separations that partitions the whole BN but find the d-separation that can partition the subgraph. The subgraph may contain suitable one-vertex d-separations. For the example of subnetwork "WIN95PTS(Problem1)" which is formed with vertex "Problem1" and all its ancestors (36 vertexes in total). Partitioning in this subnetwork brings significant improvement in both ZDD size and inference time than conventional ZDD-based method as shown in Table 5.3. Also, for the subnetwork "BARLEY(ngtilg)", though we could not generate ZDDs for the whole BARLEY, we still can query some of the vertexes by constructing the subnetwork.

5.5 Experiment Results for Related Works

The Minimum fill-in method (min-fill) and minimum degree method (min-degree) are known as the most two famous heuristic methods for the triangulation to construct a junction tree [14]. The min-fill is to eliminate the variable that leads to adding the smallest number of fill-in edges. The min-

degree is to eliminate the variable that has the smallest number of neighbors in the graph. We present the conventional jointree based method results using *min-fill* and *min-degree* in Table 5.6 and Table 5.7 . According to this table, the on-line inference using ZDD-based method works much better than jointree based methods as long as we are able to generate ZDDs for a BN. Therefore, efficiently generating ZDDs for BNs is quite valuable to be taken into account.

Another popular method to compile a BN using symbolic logics is the CNF-based compiling method [13]. In their method, BNs is encoded into Conjunctive Normal Forms (CNFs) which then can be factored and compactly represented by arithmetic circuits. Evaluating and Differentiating the arithmetic circuits solves the exact inference efficiently. Their method also provides expressive frameworks for exploiting local structure and known as one of the most efficient methods that compiles BN with Decision Diagrams. Our method does not use the CNF representation but directly translates a BN into a set of factored MLFs using ZDDs. Table 5.8 is the results of their method using the same dataset in Table 5.1. For the networks ALARM and HEPAR2, the ZDD-based method are competitive to theirs through partitioning. Also, our idea of partitioning BNs is independent of data structures so that we hope it would also fit their method.

5.6 Summary

We proposed a method of divide-and-conquer that partitions BNs into conditionally independent components using one-vertex serial pattern d-separations and separately compile these components. Through the partitioning, we get much smaller ZDDs and largely reduce time for compilation than the conventional ZDD-based method if there exists suitable one-vertex

d-separations in given BNs. Through our experiments, we know that we can partition a given BN with all one-vertex d-separations without worrying about reducing the efficiency of compiling and inference.

In Chapter 4, we discussed the method to separately compile a BN by compiling a given jointree into ZDDs and perform the message passing algorithm on ZDDs. However, straightforwardly compiling a jointree into ZDDs results in a large number of components and message variables so that the time consumption for compiling and inference both are unacceptable. In this chapter, through restricting the d-separations size to one-vertex, we avoid partitioning a BN into too many components and using too many message variables. Thus,the inference efficiency is guaranteed.

As a future work, we hope to use d-separations consisting of more than one vertex to partition a BN and inspecting the trade-off between compiling and inference to further improve the ZDD-based method for BN inference.

Chapter 6

Conclusions and Open Problems

In this chapter, we conclude our contributions presented in this thesis. We also give open problems for our proposed methods and future works for this thesis.

6.1 Concluding Remarks

The exact inference problem in BNs is known NP hard and kinds of efforts to accelerate the inference have been put forward to increase the practicality of BNs. In this thesis, we presented methods using ZDD-based logic operations to accelerate the exact inference. Our works mainly deal with improving the conventional ZDD-based inference method from two factors, to condense ZDD size through factorizations and to compile decomposition forms

of BNs instead of the whole networks.

In Chapter 3, we discussed how to condense the size of ZDDs in the conventional ZDD-based method. We proposed to factorize a large ZDDs into several small ones using the fast weak division algorithm in ZDDs. We use the ZDDs of manually d-separation structures as divisors to factorize a large ZDD. Through these factorizations, the ZDD size is largely condensed and time for factorizations are significantly reduced comparing with the conventional factorization algorithm. In Chapter 4, we considered to straightly compile a junction tree, one of the most prevalent decomposition forms of BN, using ZDDs. Through introducing message variables into ZDDs, we can perform the popular message passing algorithm on ZDDs just as it is performed on a junction tree. Inference is largely accelerated though local structure exploiting and cache memory techniques. The efficiency of this approach hints us that decomposing a BN to reduce the BN scale would bring large improvement for exact inference. But experiments that failed to compile a junction tree into ZDDs also alerts us that decomposing a BN into too many components would give rise to a blow up of ZDD size. Correspondingly, in Chapter 5, we proposed to partition and separately compile BNs using ZDDs. We use all one-vertex d-separation sets to partition a BN into several conditionally independent components. Then these components are separately compiled into node-sharing ZDDs. By restricting to one-vertex d-separations, we avoid of partitioning a BN into too many components and ensured the efficiency of exact inference. Though this method, not only size of ZDDs are significantly condensed comparing with the conventional ZDD-based method, but also exact inference are largely accelerated.

With above three propositions, we can establish a systematic strategy

of accelerate exact inference of BNs using any other arithmetic circuits since they are also suitable for any other logic based compilation method. Firstly, using some vertex sets in a BN to partition a large scale of BN into several small components. These vertex sets with these components should satisfy the independence assertions in BNs to make sure the validity of synthesis calculation results from every component. Secondly, compile these small components into arithmetic circuits and design inference algorithms capable to answer any given queries on the circuits. Exploiting local structures and cache memory techniques are two main advantages of logic based compilation method. Not only BNs can be compactly represented but also exact inference can be accelerated significantly which breaks the barrier of treewidth bottleneck in exact inference of BNs.

6.2 Open Problems and Future Directions

As future directions, there are still several challenges. Firstly, a precise trade-off between the number of components and time of inference need to be taken into account. In this thesis, we use all the one-vertex d-separation vertex sets to partition a BN. As shown in the experiment results, the ZDD size is further reduced if we partition a BN with two-vertex d-separations. Thus, it is valuable to discuss partitioning with two- or more than two vertexes d-separations and providing a practical constraints to find the most appropriate d-separations so that the inference time can be accelerated, too. Secondly, a fast algorithm to find appropriate d-separations in a BN need to be considered. These d-separations should satisfy constrains of partitioning a BN into several balanced components meanwhile the total number of instantiations over d-separations is minimum. As far as the author know, there are some approaches

to find a minimum d-separation given a BN using the maxflow-mincut based method. There are also some algorithms to find a cut vertex set for a graph to partition it into two balanced components. But methods satisfying both above two constraints have not been touched before. In Chapter 3, we try to find d-separations manually to factorize a large ZDD. In Chapter 5, we utilize Tarjan's cut vertex finding algorithm to find all one-vertex d-separations. We hope that a convenient and fast algorithm to find d-separations would bring great improvement for our method. Also, we hope to apply our method on the realistic BN data.

Acknowledgements

I would like to thank my supervisor Professor Shin-ichi Minato who not only patiently advised and taught me many valuable technical things for this research, but also encouraged and supported me all the time throughout the entire master's and doctoral courses. I also would like to thank as another supervisor Professor Hiroki Arimura for his valuable comments and encouragements for thesis and also his help for the relative matters for my graduation. I would like to specially thank Professor Thomas Zeugmann as an advisor who advised me the knowledge I should have and how to do a presentation as a doctor. I also would like to thank Associate Professor Ichigaku Takigawa as an advisor for his valuable comments and encouragements for thesis. I would like to thank my co-authors, especially, Dr. Masakazu Ishihata for his meaningful discussion and advice. He spared no efforts to show me how to conduct research, what a researcher should be like, and what kind of a philosophy I should have. I would like to thank the secretaries, Mis Sachiko Soma from Large-Scale Knowledge Processing Laboratory, Mis Yu Manabe from Information Knowledge Network Laboratory, Mis Mamie Abe from Algorithm Laboratory and Mis Yukie Watanabe from JSPS KAKENHI KIBAN (S) Discrete Structure Manipulation System Project . They supported me in many complicated office tasks of the university. I also would like to thank all the students in these laboratories who help me a

lot in both my research and my Japanese studying. I also would like to thank the China Government CSC project which provided financial support for my fist four years life of study aboard which allowed me to focus on my research without worrying about money.

Finally, I would like to express my special thanks to my mother Yanhong Gu and my father Guirong Gao being patient and having understanding during my six years life abroad in the absence of company from their only child.

Related Publications

[a] Shan Gao, Masakazu Ishihata and Shin-ichi Minato, Separate compilation of Bayesian networks for efficient exact inference *Artificial Intelligence*, volume 33, number 6A, The Japanese Society of Artificial Intelligence (JSAI), 2018. (to appear)

[b] Shan Gao, Masakazu Ishihata and Shin-ichi Minato. Fast message passing algorithm using ZDD-based local structure compilation. *in Proceedings of the 3rd Workshop on Advanced Methodologies for Bayesian Networks*, AMBN2017, Proceedings of Machine Learning Research, PMLR, pages 117–128, 2017.

[c] Shan Gao, Shin-ichi Minato. Factorization of ZDDs for representing Bayesian networks based on d-separations, In *In Workshop on Advanced Methodologies for Bayesian Networks*, AMBN 2015, Proceedings, Lecture Notes in Artificial Intelligence, Springer, volume 9505, pages 168–183, 2015

Copyright Notice

Bibliography

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, volume 27, number 6, pages 509–516, 1978.

[2] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Value elimination: Bayesian inference via backtracking search. In Uffe Kjærulff and Christopher Meek, editors, *UAI'03: Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 7-10*, pages 20–28. Morgan Kaufmann, 2003.

[3] Patrick Billingsley. *Probability and Measure*, volume 939. John Wiley & Sons, 2012.

[4] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *UAI'96: Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4*, pages 115–123. Morgan Kaufmann, 1996.

[5] Guy Bresler, Elchanan Mossel, and Allan Sly. Reconstruction of Markov random fields from samples: Some observations and algorithms. In Ashish Goel, editor, *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 11th International Workshop, APPROX 2008, and 12th International Workshop, RANDOM 2008,*

Boston, MA, USA, August 25-27, Proceedings, volume 5171 of *Lecture Notes in Computer Science*, pages 343–356. Springer, Berlin, Heidelberg, 2008.

[6] Marcos Chavira. *Beyond treewidth in probabilistic inference*. PhD thesis, University of California, Los Angeles, 2007.

[7] Mark Chavira, David Allen, and Adnan Darwiche. Exploiting evidence in probabilistic inference. In Fahiem Bacchus and Tommi Jaakkola, editors, *UAI'05: Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29*, pages 112–119. AUAI Press, 2005.

[8] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, Scotland, July 30-August 5*, pages 1306–1312. Morgan Kaufmann, 2005.

[9] David Maxwell Chickering, David Heckerman, and Christopher Meek. A Bayesian approach to learning Bayesian networks with local structure. In Prakash Shenoy Dan Geiger, editor, *UAI'97: Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence, Brown University, Providence, Rhode Island, USA, August 1-3*, pages 80–89. Morgan Kaufmann, 1997.

[10] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, volume 48, number 4, pages 608–647, 2001.

[11] Adnan Darwiche. A logical approach to factoring belief networks. In Dieter Fensel, Fausto Guinchiglia, Deborah McGuinness, and Mary-AnneWilliams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25*, pages 409–420. Morgan Kaufmann, 2002.

[12] Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, volume 50, number 3, pages 280–305, 2003.

[13] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, Valencia, Spain, August 22-27*, pages 328–332. IOS Press, 2004.

[14] Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

[15] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, volume 38, number 3, pages 353–366, 1989.

[16] William Feller. *An introduction to probability theory and its applications*, volume 1. Wiley, New York, 1968.

[17] Brendan J. Frey and Nebojsa Jojic. A comparison of algorithms for inference and learning in probabilistic graphical models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 27, number 9, pages 1392–1416, 2005.

[18] Nir Friedman and Moises Goldszmidt. Learning Bayesian networks with local structure. In *UAI '96: Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4*, pages 252–262. Morgan Kaufmann, 1996.

[19] Shan Gao, Masakazu Ishihata, and Shin-ichi Minato. Fast message passing algorithm using ZDD-based local structure compilation. In Antti Hyttinen, Joe Suzuki, and Brandon Malone, editors, *Proceedings of the 3rd Workshop on Advanced Methodologies for Bayesian Networks, AMBN2017, Kyoto, Japan, September 20-22, Proceedings of Machine*

Learning Research, volume 73, pages 117–128. PMLR, 2017.

[20] Dan Geiger, Thomas Verma, and Judea Pearl. Identifying independence in Bayesian networks. *Networks*, volume 20, number 5, pages 507–534, 1990.

[21] David Gregory, Karen Bartlett, Aart deGeus, and Gary Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. In *Papers on 25 years of electronic design automation*, pages 580–586. ACM, 1988.

[22] Anders Hald. *A history of probability and statistics and their applications before 1750*, volume 501. John Wiley & Sons, 2003.

[23] HebrewUniversity. Bayesian network repository. `http://www.cs.huji.ac.il/~galel/Repository/`.

[24] David Heckerman. Bayesian networks for data mining. *Data Mining and Knowledge Discovery*, volume 1, number 1, pages 79–119, 1997.

[25] David Heckerman. A tutorial on learning with Bayesian networks. *Nato Asi Series D Behavioural And Social Sciences*, volume 89, pages 301–354, 1998.

[26] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, volume 15, number 3, pages 225–263, 1996.

[27] Manfred Jaeger. On the complexity of inference about probabilistic relational models. *Artificial Intelligence*, volume 117, number 2, pages 297–308, 2000.

[28] Finn V. Jensen. *Introduction to Bayesian Networks*. Springer-Verlag New York, Inc., 1996.

[29] Olav Kallenberg. *Foundations of modern probability*. Springer Science & Business Media, 2006.

[30] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In Giuseppe De Giacomo Chitta Baral and Thomas Eiter, editors, *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR-14), Vienna, Austria, July 20-24*, pages 1–10. AAAI, 2014.

[31] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques-Adaptive Computation and Machine Learning*. MIT Press, 2009.

[32] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, volume 47, number 2, pages 498–519, 2001.

[33] Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, volume 50, number 2, pages 157–224, 1988.

[34] Chao Li and Maomi Ueno. A fast clique maintenance algorithm for optimal triangulation of Bayesian networks. In Joe SuzukiMaomi Ueno, editor, *Workshop on Advanced Methodologies for Bayesian Networks, AMBN 2015, Yokohama, Japan, November 16-18,proceedings, Lecture Nodes in Artificial Intelligence, Springer*, volume 9505, pages 152–167. Springer, Cham, 2015.

[35] Chao Li and Maomi Ueno. An extended depth-first search algorithm for optimal triangulation of Bayesian networks. *International Journal of Approximate Reasoning*, volume 80, number C, pages 294–312, 2017.

[36] Anders L. Madsen and Finn V. Jensen. Lazy propagation in junction trees. In Gregory Cooper and Serafin Moral, editors, *Proceedings of the 14th conference on Uncertainty in Artificial Intelligence, Madison, Wis-*

consin,July 24-26, pages 362–369. Morgan Kaufmann, 1998.

[37] Ole J. Mengshoel. Understanding the scalability of Bayesian network inference using clique tree growth curves. *Artificial Intelligence*, volume 174, number 12-13, pages 984–1006, 2010.

[38] Shin-ichi Minato. Fast weak-division method for implicit cube representation. In *Synthesis And System Integration of Mixed Information Technologies(SASIMI), October 20-22, Nara, Japan*, volume 93, pages 423–432, 1993.

[39] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference, Dallas, TX, USA, June 14-18*, pages 272–277. ACM, 1993.

[40] Shin-ichi Minato. Fast factorization method for implicit cube set representation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 15, number 4, pages 377–384, 1996.

[41] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, volume 3, number 2, pages 156–170, 2001.

[42] Shin-ichi Minato, Ken Satoh, and Taisuke Sato. Compiling Bayesian networks by symbolic probability calculation based on Zero-suppressed BDDs. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI2007), Hyderabad, India, January 06-12*, pages 2550–2555. Morgan Kaufmann, 2007.

[43] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI '99: Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30-August 1*, pages 467–475. Morgan

Kaufmann, 1999.

[44] Nasser M. Nasrabadi. Pattern recognition and machine learning. *Journal of Electronic Imaging*, volume 16, number 4, pages 049901, 2007.

[45] Richard E. Neapolitan. *Learning Bayesian networks*. Pearson Prentice Hall Upper Saddle River, NJ, 2004.

[46] Gregory Nuel. Tutorial on exact belief propagation in Bayesian networks: from messages to algorithms. *ArXiv Preprint ArXiv:1201.4724*, 2012.

[47] Thorsten J. Ottosen and Jirí Vomlel. All roads lead to rome - new search methods for the optimal triangulation problem. *International Journal of Approximate Reasoning*, volume 53, number 9, pages 1350–1366, 2012.

[48] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.

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

[50] Scott Sanner and David McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In Menno van Zaanen, Tim Oates, Georgios Paliouras, and Colin de la Higuera, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence(IJCAI-05), July 30-August 5, Edinburgh, Scotland*, pages 1384–1390. Morgan Kaufmann, 2005.

[51] Luis Enrique Sucar. *Probabilistic Graphical Models: Principles and Applications*. Springer Publishing Company, Incorporated, 2015.

[52] Robert Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, volume 2, number 6, pages 160–161, 1974.

[53] Wim Wiegerinck. Variational approximations between mean field theory and the junction tree algorithm. In Craig Boutilier and Moisés Gold-

szmidt, editors, *UAI'00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, Stanford, California, June 30-July 03*, pages 626–633. Morgan Kaufmann, 2000.

[54] Dan Wu and Cory Butz. On the complexity of probabilistic inference in singly connected Bayesian networks. In Dominik Ślęzak, JingTao Yao, James F. Peters, Wojciech Ziarko, and Xiaohua Hu, editors, *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing, 10th International Conference, RSFDGrC 2005, Proceedings, Regina, Canada, August 31-September 3, Part I*, pages 581–590. Springer, Berlin, Heidelberg, 2005.