2016

Structure Discovery in Bayesian Networks: Algorithms and Applications

Yetian Chen
Iowa State University

Structure discovery in Bayesian networks: Algorithms and applications

by

Yetian Chen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Jin Tian, Major Professor

Kris De Brabanter

David Fernández-Baca

Yan-Bin Jia

Daniel S. Nettleton

Iowa State University

Ames, Iowa

2016

DEDICATION

I would like to dedicate this thesis to my parents without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to many people without whose support and help I would not have been able to complete this dissertation.

First and foremost, I would like to thank my advisor, Dr. Jin Tian for his guidance and support throughout the years of my PhD studies. I want to thank him for giving me the freedom to choose the research topics that I am most interested in while keeping me going in the right direction, and for the guidance on developing my research methodology and career. His insights, patience and focus have been unparalleled and have inspired me to contribute all that I have to the pursuit of knowledge. I am also greatly thankful to my committee members, Drs. Kris De Brabanter, David Fernández-Baca, Yan-Bin Jia and Daniel S. Nettleton for all their support and advice.

I would like to thank Dr. Kewei Tu and Yanpeng Zhao at Shanghai Tech University for the discussions and collaborations in the research of curriculum learning of Bayesian networks. Thanks to Dr. Srinivas Aluru at Georgia Tech and Dr. Olga Nikolova for their work, discussions and collaborations in the research of parallel algorithms for learning graphical models. I would also like to thank Ru He for his thesis work and discussions that partly motivated my current research.

I would like to thank my colleagues during my research internship at Pearson Research & Innovation Network, especially José P. González-Brenes, John Behrens, Johann Ari Larusson, Tom McTavish, Ilya Goldin, J.D. Corbin and Tasmin K. Dhaliwal for their research and discussions with me that motivated my work of using Bayesian networks for student modeling. A special thank to José P. González-Brenes for introducing me to the field of educational data mining and personalized learning.

I would like to thank all my friends that I made before and during my stay at Ames. I would especially thank Feng Guo, Yueran Yang, Yang Peng, Xiang Huang, Qiuyan Liao, Chuan Jiang,

Wangyujue Hong, Sen Wang, Wei Zhang and Lisen Peng for their tremendous help, support and friendship. Without them, Ames wouldn't be such a lovely place to me.

Finally, my deepest thanks to my parents and my girlfriend Yu Liu for their unconditional love and support: thank you for understanding and supporting my choice of pursuing PhD abroad, and for tolerating my staying in school for so long: I am graduating at last :)

ABSTRACT

Bayesian networks are a class of probabilistic graphical models that have been widely used in various tasks for probabilistic inference and causal modeling. A Bayesian network provides a compact, flexible, and interpretable representation of a joint probability distribution. When the network structure is unknown but there are observational data at hand, one can try to learn the network structure from the data. This is called structure discovery.

Structure discovery in Bayesian networks is a host of several interesting problem variants. In the optimal Bayesian network learning problem (we call this *structure learning*), one aims to find a Bayesian network that best explains the data and then utilizes this optimal Bayesian network for predictions or inferences. In others, we are interested in finding the local structural features that are highly probable (we call this *structure discovery*). Both *structure learning* and *structure discovery* are considered very hard because existing approaches to these problems require highly intensive computations.

In this dissertation, we develop algorithms to achieve more accurate, efficient and scalable structure discovery in Bayesian networks and demonstrate these algorithms in applications of systems biology and educational data mining. Specifically, this study is conducted in five directions.

First of all, we propose a novel heuristic algorithm for Bayesian network *structure learning* that takes advantage of the idea of *curriculum learning* and learns Bayesian network structures by stages. We prove theoretical advantages of our algorithm and also empirically show that it outperforms the state-of-the-art heuristic approach in learning Bayesian network structures.

Secondly, we develop an algorithm to efficiently enumerate the k-best equivalence classes of Bayesian networks where Bayesian networks in the same equivalence class are equally expressive in terms of representing probability distributions. We demonstrate our algorithm in the task of Bayesian model averaging. Our approach goes beyond the maximum-a-posteriori (MAP)

model by listing the most likely network structures and their relative likelihood and therefore has important applications in causal structure discovery.

Thirdly, we study how parallelism can be used to tackle the exponential time and space complexity in the exact Bayesian structure discovery. We consider the problem of computing the exact posterior probabilities of *modular structural features*, e.g., directed edges, in Bayesian networks. We present a parallel algorithm capable of computing the exact posterior probabilities of all possible directed edges with optimal parallel space efficiency and nearly optimal parallel time efficiency. We apply our algorithm to a biological data set for discovering the *yeast* pheromone response pathways.

Fourthly, we develop novel algorithms for computing the exact posterior probabilities of ancestor relations (*non-modular features*) in Bayesian networks. Existing algorithm assumes an order-modular prior over Bayesian networks that does not respect Markov equivalence. Our algorithm allows uniform prior and respects the Markov equivalence. We apply our algorithm to a biological data set for discovering protein signaling pathways.

Finally, we introduce *Combined student Modeling and prerequisite Discovery* (COMMAND), a novel algorithm for jointly inferring a prerequisite graph and a student model from student performance data. COMMAND learns the skill prerequisite relations as a Bayesian network, which is capable of modeling the global prerequisite structure and capturing the conditional independence between skills. Our experiments on simulations and real student data suggest that COMMAND is better than prior methods in the literature. COMMAND is useful for designing intelligent tutoring systems that assess student knowledge or that offer remediation interventions to students.

CHAPTER 1. INTRODUCTION

Probabilistic graphical models (PGMs) use a graph-based representation to compactly encode a joint distribution by making conditional independence (CI) assumptions. In particular, the nodes in the graph represent random variables, and the (lack of) edges represent CI assumptions. There are several types of graphical model, depending on whether the graph is composed of directed, undirected, or some combination of directed and undirected edges. In this dissertation, we mainly focus on directed graphical models, i.e., Bayesian networks.

A Bayesian network is a directed acyclic graph (DAG) where each node and its parents are associated with a conditional probability distribution (CPD). A CPD quantifies the effect of the parents on the node. Figure 1.1 provides an example of Bayesian network that models the probability dependence among four binary random variables: *Cloudy*, *Sprinkler*, *Rain* and *WetGrass*. These models are also called belief networks, or sometimes, causal networks, because the directed edges are sometimes interpreted as causal relations. For example, in Figure 1.1, the directed edge between *Rain* and *WetGrass* can be interpreted as that raining could cause wet grass. Bayesian networks have been widely used in various tasks for probabilistic inference and causal modeling (Pearl, 2000; Spirtes et al., 2000). The DAG structure and the associated CPDs of a Bayesian network can be constructed manually using the domain knowledge. However, in many applications, we lack enough domain knowledge and have to learn the structure as well as the CPDs from the data.

Learning a Bayesian network is often conducted in two phases. First, one learns the DAG structure. In the second phase, one estimates the parameters of the conditional distributions given the fixed structure. Parameter estimation in the second phase is considered a well-studied problem. The learning of the DAG structure, or in other words, structure learning, is more challenging.

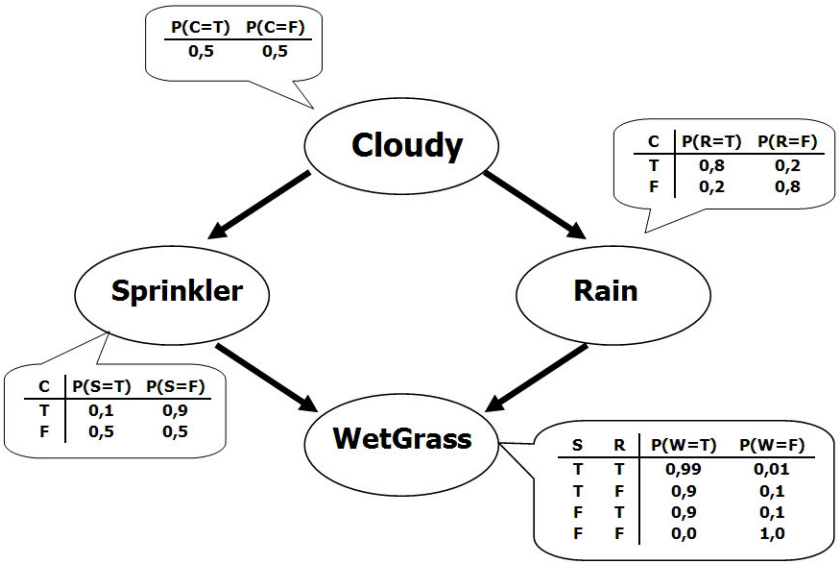


Figure 1.1: An example of Bayesian network with four variables.

Structure discovery in Bayesian networks is a host of several interesting problem variants. In the optimal Bayesian network learning problem (we will call it *structure learning*), one aims to find a Bayesian network that best explains the data and then uses this optimal Bayesian network for predictions or inferences. This problem is also called model selection in relevant literature.

In another problem variant, we are interested in finding the highly probable local structural features (we will call this *structure discovery*) instead of identifying the overall structure of the Bayesian network. For example, a directed edge in a Bayesian network represents direct causal relation between two variables; a directed path, composed of consecutively directed edges, represents (indirect) causal relation between two variables; a Markov blanket (MB) of a variable, composed of its parents, children and spouses (children's parents), shielding the node from the rest of the network, is the only knowledge needed to predict the behavior of that variable (Pearl, 1988). Learning these structural features from data is of great interest.

Both *structure learning* and *structure discovery* are considered very challenging because existing approaches to these problems require highly intensive computations as well as large memory usages. In this thesis, we aim to develop algorithms to achieve more accurate, efficient and scalable structure learning and structure discovery in Bayesian networks.

1.1 Preliminaries and Problem Definition

In this section, we present some preliminaries of Bayesian networks and provide the problem definition for structure learning and structure discovery in Bayesian networks.

1.1.1 Bayesian Networks

A Bayesian network is a pair $B = (G, P)$, where G is a DAG that encodes a joint probability distribution P over a vector of random variables $\mathbf{X} = (X_1, ..., X_n)$ with each node of the graph representing a variable in $\mathbf{X}$. In this dissertation, we will use random variable and node interchangeably. For convenience we typically work on the index set $V = \{1, ..., n\}$ and represent a variable X_i by its index i. The DAG can be represented as a vector $G = (Pa_1, ..., Pa_n)$ where each Pa_i is a subset of $V \setminus \{i\}$ and specifies the parents of X_i in the graph. Each node and its parents in the DAG is associated with a conditional probability distribution (CPD) $P(X_i|Pa_i)$. Then the joint distribution $P(\mathbf{X})$ must be factorized as follows:

$$P(\mathbf{X}) = \prod_{i=1}^{n} P(X_i|Pa_i). \tag{1.1}$$

Equation 1.1 is called the chain rule of Bayesian network.

Definition 1.1 (Conditional Independence (CI))**.** *Let* $\mathbf{X}$*,* $\mathbf{Y}$ *and* $\mathbf{Z}$ *be three disjoint sets of random variables. We say that* $\mathbf{X}$ *is conditionally independent of* $\mathbf{Y}$ *given* $\mathbf{Z}$*, denoted by* $I(\mathbf{X}, \mathbf{Z}, \mathbf{Y})$*, if for any values* $\mathbf{x}, \mathbf{y}, \mathbf{z}$ *of* $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ *where* $P(\mathbf{Z} = \mathbf{z}) > 0$*,*

$$P(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y}|\mathbf{Z} = \mathbf{z}) = P(\mathbf{X} = \mathbf{x}|\mathbf{Z} = \mathbf{z})P(\mathbf{Y} = \mathbf{y}|\mathbf{Z} = \mathbf{z}).$$

A DAG G encodes a set of conditional independence (CI) relations over the variable set $\mathbf{X}$. These CI relations can be determined using a graphical criterion called *d-separation* (Pearl, 1988), which is defined on the basis of blocked paths.

A path between two nodes X_i and X_j in a DAG G consists of a sequence of consecutive edges (ignoring the direction). A node X_i is said to be an ancestor of a node X_j if there is a directed path $X_i \to \cdots \to X_j$. X_j is called a descendant of X_i. A non-endpoint node Y on a path is called a collider if two arrowheads on the path meet at Y, i.e.,$\to Y \leftarrow$; all other non-endpoint nodes on a path are non-colliders, i.e., $\leftarrow Y \to$, $\leftarrow Y \leftarrow$ and $\to Y \to$.

Definition 1.2 (d-separation)**.** *(Pearl, 1988) A path between nodes X_i and X_j in a DAG G is said to be d-separated (or blocked) by a set of nodes $\mathbf{Z}$ if and only if*

1. *there is a non-collider on the path in $\mathbf{Z}$, or*

2. *there is a collider not in $\mathbf{Z}$ and none of this collider's descendants is in $\mathbf{Z}$.*

X_i and X_j are said to be d-separated given $\mathbf{Z}$, denoted by $dsep_G(X_i, \mathbf{Z}, X_j)$, if every path between X_i and X_j is d-separated or blockd by $\mathbf{Z}$.

The concept of *d-separation* is illustrated in Figure 1.2. In this example, $dsep_G(X_1, \{X_4, X_5\}, X_7)$ holds because $\{X_4, X_5\}$ blocks the only path between X_1 and X_7 and neither X_4 nor X_5 is a collider. However, neither $dsep_G(X_1, \{X_2\}, X_7)$ nor $dsep_G(X_1, \{X_6\}, X_7)$ holds because both X_2 and X_6 are colliders.

D-separation can be generalized for sets of nodes, that is, sets $\mathbf{X}$ and $\mathbf{Y}$ are said to be d-separated given $\mathbf{Z}$, denoted by $dsep_G(\mathbf{X}, \mathbf{Z}, \mathbf{Y})$, if for every pair X_i, Y_j, with $X_i \in \mathbf{X}$, $Y_j \in \mathbf{Y}$, X_i and Y_j are d-separated given $\mathbf{Z}$.

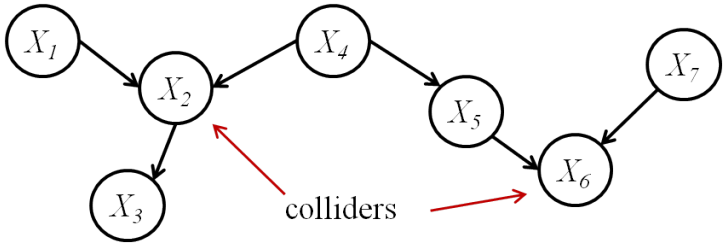


Figure 1.2: An illustrative example of *d-separation*.

The set of all the conditional independence (CI) relations encoded by a DAG G is specified by the following global Markov property.

Definition 1.3 (The Global Markov Property (GMP))**.** *A probability distribution P is said to satisfy the global Markov property for G if for any disjoint sets $\mathbf{X} \neq \emptyset$, $\mathbf{Y} \neq \emptyset$, $\mathbf{Z}$,*

$$dsep_G(\mathbf{X}, \mathbf{Z}, \mathbf{Y}) \Longrightarrow I(\mathbf{X}, \mathbf{Z}, \mathbf{Y}).$$

We define $\mathcal{I}(G) = \{I(\mathbf{X}, \mathbf{Z}, \mathbf{Y}) : dsep_G(\mathbf{X}, \mathbf{Z}, \mathbf{Y})\}$, i.e., the set of all CIs implied by the global Markov property of a DAG G.

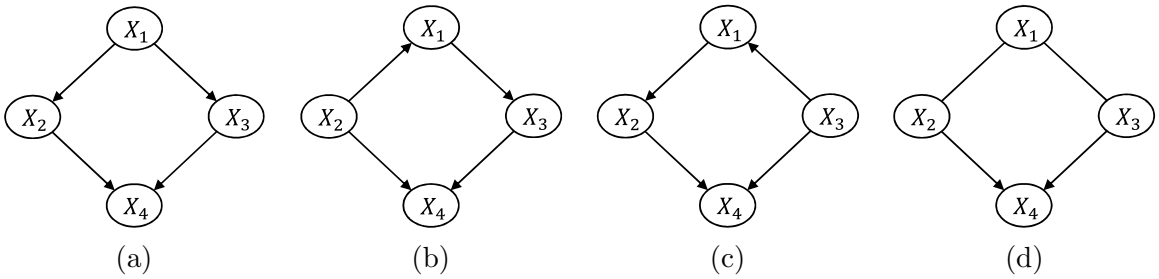


Figure 1.3: An equivalence class containing three DAGs (a, b, c) and its CPDAG (d).

The Global Markov Property says for a Bayesian network $B = (G, P)$, every *d-separation* in the DAG G implies a CI relation that must hold in P.

Definition 1.4 (I-equivalence)**.** *(Verma and Pearl, 1990) Two DAG G_1 and G_2 over the same set of variables $\mathbf{X}$ are I-equivalent if $\mathcal{I}(G_1) = \mathcal{I}(G_2)$, i.e., they represent the same set of CI relations. The set of all DAGs over $\mathbf{X}$ are partitioned by the I-equivalence relationship into a set of mutually exclusive and exhaustive equivalence classes (ECs).*

Two I-equivalent DAGs are statistically indistinguishable. That is, given observational data, it is impossible to identify a unique data-generating DAG unless there is only one DAG in the corresponding equivalence class. This property has substantial impact on both structure learning and structure discovery.

Equivalent DAGs have some common structural features. DAGs in the same equivalence class (EC) have the same skeleton and the same v-structures[1] (Verma and Pearl, 1990). Thus, we can represent an EC using a complete partially DAG (CPDAG) which consist of a directed edge for every irreversible edge and an undirected edge for every reversible edge.[2] Figure 1.3 shows an example equivalence class containing three DAGs and the corresponding CPDAG (Figure 1.3d).

Definition 1.5 (Independencies in P)**.** *Let P be a distribution over $\mathbf{X}$. We define $\mathcal{I}(P)$ to be the set of all independence assertions of the form $I(\mathbf{X}, \mathbf{Z}, \mathbf{Y})$ that hold in P.*

Definition 1.6 (Perfect Map)**.** *We say a DAG G is a perfect map (P-map) for a distribution P if $\mathcal{I}(G) = \mathcal{I}(P)$.*

[1] A v-structure in a DAG G is an ordered triple of nodes (u, v, w) such that G contains the directed edges $u \to v$ and $w \to v$ and u and w are not adjacent in G.

[2] A CPDAG is also called a *pattern*. Each equivalence class has a unique CPDAG.

A Bayesian network $B = (G, P)$ where G is a perfect map of P is called a faithful Bayesian network (Spirtes et al., 2000). Not every distribution has a perfect map. However, these distributions are "rare" (Meek, 1995). In the problem of Bayesian network structure learning, we will assume the distribution P has a P-map (may not be unique) and our goal is to find a perfect map for P.

1.1.2 Structure Learning of Bayesian Networks

In the problem of Bayesian network structure learning, one aims to find a Bayesian network that best explains the observed data. More formally, we assume that the data D are generated i.i.d from an underlying distribution $P^*(\mathbf{X})$ which is induced by some Bayesian network $B^* = (G^*, P^*)$. Our goal is to find a (the) perfect map G^* for P^*. Due to the so called I-equivalence, the best we can hope for is to recover G^*'s equivalence class. That is, we target any G that is I-equivalent to G^*.

In general, there are two main approaches for learning Bayesian networks from data. The first one is constraint-based (Spirtes et al., 2000). Algorithms following this approach estimate from the data whether certain conditional independencies (CIs) between the variables hold. The CI constraints are propagated throughout the graph and the DAGs that are inconsistent with them are eliminated from further consideration. A sound strategy for performing CI tests ultimately retains (and returns) only the I-equivalent DAGs consistent with the tests.

The other approach is score-based search that converts the learning problem to an optimization problem. Algorithms following this approach attempt to optimize a scoring function that measures how well a DAG fits the data (Cooper and Herskovits, 1992; Heckerman et al., 1995). In this dissertation, we focus on the score-based search algorithms.

The first component of the score-and-search method is a scoring criterion measuring the fitness of a DAG G to the data D. Several commonly used scoring functions are MDL, AIC, BIC and Bayesian score. In this work, we use Bayesian score, defined as follows:

$$score(G : D) = \log P(D|G) + \log P(G), \tag{1.2}$$

where $P(D|G)$ is the likelihood of the data given the DAG G and $P(G)$ is the prior over the

DAG structures. Assuming global and local parameter independence, parameter modularity, and uniform structure prior $P(G)$, the score is decomposable (Heckerman et al., 1995):

$$score(G : D) = \sum_{i=1}^{n} score_i(Pa_i : D), \tag{1.3}$$

where $score_i(Pa_i : D)$ is called the local score or family score measuring how well a set of variables Pa_i serves as parents of X_i. It is desirable that for any two I-equivalent DAGs G_1 and G_2, $score(G_1 : D) = score(G_2 : D)$. This is called score equivalence. The commonly used scoring functions such as MDL, AIC, BIC and BDe all satisfy score decomposability and equivalence.

In this work, we assume discrete random variables that follow a Dirichlet-Multinomial distribution. That is, each variable X_i follows a multinomial distribution with parameter vector Θ_{i,Pa_i} conditioning on its parents, and the parameter vector Θ_{i,Pa_i} follows a Dirichlet distribution with hyperparameter vector α_{i,Pa_i} as the prior. In this thesis we use the following BDeu score with $\alpha_{i,Pa_i} = N'/r_i q_i$ (Buntine, 1991):

$$score_i(Pa_i : D) = \sum_{j=1}^{q_i} \left[\ln \left(\frac{\Gamma(\frac{N'}{q_i})}{\Gamma(N_{ij} + \frac{N'}{q_i})} \right) + \sum_{k=1}^{r_i} \ln \left(\frac{\Gamma(N_{ijk} + \frac{N'}{r_i q_i})}{\Gamma(\frac{N'}{r_i q_i})} \right) \right], \tag{1.4}$$

where r_i is the number of possible states of variable X_i; q_i is the number of possible configurations of the parent set Pa_i of X_i; N_{ijk} is the number of instances in data D where the variable X_i takes its k-th value x_{ik} and the variables in Pa_i take their j-th configuration; $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$; and N' is called the equivalent sample size representing the strength of our belief in the prior distribution of parameters. The BDeu score can be computed efficiently from the sufficient statistics of the data D.

Given a scoring function, the goal of the search procedure is to identify a best scoring DAG by searching in the space of all possible DAGs. Since the size of the DAG space is of the order $O(n!2^{n(n-1)/2})$ with respect to the number of nodes n, the search problem is considered very hard. Indeed, it has been proved that finding a best Bayesian network is NP-hard when using the BDeu scoring criterion (Chickering, 1996).

1.1.3 Structure Discovery in Bayesian Networks

A common solution to identifying highly probable local structural features is to use Bayesian approach. Given a set of observations D, in the Bayesian approach to learn Bayesian networks from the observations, we compute the posterior probability of a DAG G by

$$P(G|D) = \frac{P(D|G)P(G)}{P(D)}, \tag{1.5}$$

where $P(G)$ is called the structure prior, $P(D|G)$ is the likelihood of the data, and $P(D)$ is the marginal probability of the data D.

A structural feature, e.g., an edge or a directed path, is conveniently represented by an indicator function f such that $f(G)$ is 1 if the feature is present in G and 0 otherwise. The posterior probability of any structural feature can be computed by averaging over all possible DAGs:

$$P(f|D) = \sum_{G} f(G)P(G|D). \tag{1.6}$$

A structural feature is said *modular* if $f(G) = \prod_{i=1}^{n} f_i(Pa_i)$, where each $f_i(Pa_i)$ is an indicator function from the subset of $V \setminus \{i\}$ to $\{0, 1\}$. In other words, the representation of a modular feature can be factorized into the product of local indicator functions. Any directed edge is a modular feature. For example, an edge $u \to v$ can be represented by setting $f_v(Pa_v) = 1$ if and only if $u \in Pa_v$, and setting $f_i(Pa_i) = 1$ for all $i \neq v$. A structural feature is *non-modular* if it is not modular, i.e, its representation cannot be factorized like the modular features. For example, a directed path from s to t (denoted by $s \rightsquigarrow t$) composed of more than one directed edges, is a non-modular feature.

Once we have the posterior probability $P(f|D)$ computed, we can make inference about the feature f based on $P(f|D)$. Thus, the key question we need answer in the problem of structure discovery is how we can accurately and efficiently compute the posterior probability $P(f|D)$.

As showed in Equation 1.6, the exact computation of these posteriors requires summation over all possible DAGs, the number of which is super-exponential with respect to the number of nodes n. Thus, exact Bayesian learning of structural features is hard in terms of both time and space requirements.

Modular features have some good properties in their representations, i.e., $f(G)$ can be factorized. This makes the computation relatively easier. Non-modular features, such as directed paths (ancestor relations), have no such property. The learning is generally considered harder.

Exact learning algorithms for structure discovery are slow and specialized for only a certain type of structural features. Alternative approaches attempt to approximate these posteriors. The central idea is to select a representative set of DAGs $\mathcal{G}$, and estimate the posterior by

$$P(f|D) \approx \frac{\sum_{G\in\mathcal{G}} P(f|G,D)P(G|D)}{\sum_{G\in\mathcal{G}} P(G|D)}. \tag{1.7}$$

With this approximation, the research problems become how we select the set of representative DAGs and how good these approximations are, i.e., how close they are to the exact posteriors.

1.2 Related Work

In this subsection, we review previous work related to Bayesian network structure learning and structure discovery.

1.2.1 Bayesian Network Structure Learning

There has been an enormous amount of work on learning Bayesian networks from data. Methods for this learning problem fall into two categories: constraint-based and score-based.

The constraint-based algorithms estimate conditional independencies in the data and build the DAGs consistent to these CIs. Typically, this estimation is performed using statistical or information theoretic measures. Well-known examples are the Peter-Clark (PC) algorithm (Spirtes et al., 2001) and the Inductive-Causation (IC) algorithm (Pearl, 2000). Other later work includes the Grow-Shrink (GS) (Margaritis and Thrun, 2000) and Total-Conditioning (TC) algorithms (Pellet and Elisseeff, 2008) that first estimate each node's Markov blanket by performing CI tests then connect nodes in a maximally consistent way. PC or IC algorithms are guaranteed to return the equivalence class that the underlying Bayesian network G^* belongs to if all the CI tests are perfect, i.e., there is no error (neither type I or type II error) in each performed CI test. Such assumption certainly does not hold in practice since any kind of

statistical test will have some probability of making errors given limited data samples. Even worse, an error of a statistical test can result in propagated errors in the consequent learning process. Thus, much research on constraint-based approach has been dedicated to improving the accuracy of CI tests (Bromberg and Margaritis, 2009), alleviating error propagation, or controlling a certain type of errors (Li and Wang, 2009).

Score-based approach converts the learning problem to an optimization problem. Algorithms following this approach attempt to maximize a scoring function that measures how well a DAG fits the data (Cooper and Herskovits, 1992; Heckerman et al., 1995). It has been proved that finding an optimal Bayesian network structure is NP-hard (Chickering, 1996). Algorithms in this category include exact algorithms that are able to find an optimal solution or heuristic algorithms that often return sub-optimal models. The research on exact algorithms started with a family of algorithms using dynamic programming (DP) (Ott et al., 2004; Koivisto and Sood, 2004; Singh and Moore, 2005; Silander and Myllymäki, 2006). These DP algorithms require exponential time and space, thus are only applicable to problems of moderate size (up to about 30 variables in current desktops). Recently, alternative approaches to finding the optimal Bayesian network have been proposed and shown being competitive or faster than the DP algorithms. These approaches include A* search (Yuan et al., 2011; Malone et al., 2011; Yuan and Malone, 2012; Malone and Yuan, 2012, 2013) and Integer Linear Programming (ILP) (Jaakkola et al., 2010; Cussens, 2011; Bartlett and Cussens, 2013). The A* search based algorithm URLearning formulates the learning problem as a shortest path finding problem and employs A* search algorithm to explore the search space (Yuan et al., 2011). ILP based algorithm GOBNILP (Globally Optimal Bayesian Network learning using ILP) casts the structure learning problem as a linear program which can be solved efficiently using existing ILP frameworks such as SCIP (Achterberg et al., 2008). GOBNILP was demonstrated to be able to handle problems with up to a few hundred variables (Bartlett and Cussens, 2013). However, GOBNILP assumes the *in-degree* (or equivalently, the number of parents) of each node is upper-bounded by a small constant.

Heuristic search method encompasses a broad class of algorithms, varying in the scoring functions being used, the search strategies being employed, and assumptions being made. The

general search strategy is, given a starting point, i.e., any DAG, by adding, deleting or reversing one or a few edges, the algorithm manages to traverse the DAG space to find a high-scoring model. As mentioned, there are super-exponential number of possible DAGs. Thus, local search strategies such as greedy or more sophisticated search algorithms are often used. The searches will often get stuck in local maxima.

Since DAGs can be grouped into a smaller set of equivalence classes (ECs) and the DAGs in the same EC are equally expressive in terms of representing probability distributions, some research proposed to search in the EC space (Madigan et al., 1996; Chickering, 2002a,b; Castelo and Kocka, 2003). The potential advantages of using the EC space instead of DAG space include: (1) In the limit of large sample size, there exists a greedy search algorithm that provably identifies a perfect map of the underlying distribution (Chickering, 2002b); (2) The cardinality of EC space is smaller than DAG space; (3) Searching in the EC space improves the efficiency of search because moves within the same EC can be avoided. The first advantage says that some theoretical guarantee can be made under the assumption of unlimited sample size. However, this assumption is too strong and does not hold in reality. The second advantage does not alleviate substantially the learning complexity either as showed in (Gillispie and Perlman, 2001) that the ratio of the number of DAGs to the number of equivalence classes reaches an asymptote around 3.7 with as few as ten nodes. Searching in the EC space may also suffer from overhead due to compulsory additional operations, e.g., converting DAGs to its equivalence class partial DAG representation and vice versa (Chickering, 2002b). Thus, although theoretically promising, in practice this strategy did not show much improvement on the simple greedy search applied to the DAG space.

Finally, ideas combining both constraint-based and score-based approaches have also been explored. A well-known algorithm is Max-Min Hill-Climbing (MMHC) algorithm (Tsamardinos et al., 2006). MMHC first estimates the parents and children (PC) set of each variable using a local discovery algorithm called $MMPC$ (Tsamardinos et al., 2003). It then performs a simple greedy hill-climbing search with the constraint that the neighbors of each variable must be in the variable's PC set. Extensive empirical evaluation has showed that MMHC outperformed on average other heuristic algorithms in terms of both the quality of reconstruction and the

computational efficiency thus it was claimed to be the current state-of-the-art. The success of MMHC builds on the idea of constraining the greedy search using the candidate PC set. However, the simple greedy search in the second phase does not provide any theoretical guarantee and would easily get stuck in local maxima.

An empirical evaluation of the impact of learning strategies on the quality of learned Bayesian networks can be found in (Malone et al., 2015).

1.2.2 Structure Discovery in Bayesian Networks

The existing methods for structure discovery in Bayesian networks can be divided into two categories: model selection approach and Bayesian approach.

Model selection approach seeks out a DAG G that maximizes certain score metric, e.g., the posterior probability $P(G|D)$ given observed data D, then infers the structures based on this single model. This is problematic because: (1) the assumed "data generating DAG" is unidentifiable from the observational data due to the so-called Markov equivalence of multiple different DAGs (Verma and Pearl, 1990); and (2) other Markov equivalence classes may fit the data almost equally well due to the noises in the data (Friedman and Koller, 2003). The latter often happens in domains where the amount of data is small relative to the size of the model. Thus, inferring the local structures based on the maximum-a-posteriori (MAP) structure may give unwarranted conclusions.

Bayesian approach circumvents the model uncertainty problem by learning the posterior distribution of these structural features (Friedman and Koller, 2003). However, exact computation of these posteriors is hard due to the super-exponentially large DAG space. Recently, a number of dynamic programming (DP) algorithms successfully reduced the computation to exponential time and space. For example, the algorithms described in (Koivisto and Sood, 2004) and (Koivisto, 2006a) can compute the exact marginal posterior probability of any modular features (e.g., an edge) and the exact posterior probabilities for all $n(n-1)$ potential edges in $O(n2^n)$ time and space, assuming that the *in-degree*, i.e., the number of parents of each node, is bounded by a constant. To deal with (harder) non-modular feature, e.g., ancestor relations, an analogous DP algorithm takes $O(n3^n)$ time and $O(3^n)$ space (Parviainen and Koivisto, 2011).

However, these algorithms require order-modular structural prior $P(G)$ and perform summation over order space instead of DAG space. As a result, the computed posteriors would bias towards DAGs compatible with more linear orders and the Markov equivalence is not respected either (Friedman and Koller, 2003). To adhere to the uniform prior, Tian and He (2009) developed a novel DP algorithm directly summing over the DAG space. This algorithm is capable of evaluating all directed edges (modular features) in $O(n3^n)$ time and $O(n2^n)$ space. But whether and how this idea can be extended to deal with non-modular features need further investigation. In one chapter of this dissertation, we will study this problem.

Exact algorithms require exponential time and space and specialize on only one certain type of structural features. Thus, much research has resorted to approximate methods. The central idea is to select a representative set of DAGs $\mathcal{G}$, and estimate the posterior by averaging over these models, i.e., $P(f|D) \approx \sum_{G\in\mathcal{G}} P(f|G,D)P(G|D)/\sum_{G\in\mathcal{G}} P(G|D)$. Among these approaches are a group of methods based on Markov Chain Monte Carlo (MCMC) technique, which provides a principled way to sample DAGs from their posterior distribution $P(G|D)$ (Madigan et al., 1995; Friedman and Koller, 2003; Eaton and Murphy, 2007; Ellis and Wong, 2008; Grzegorczyk and Husmeier, 2008; Niinimäki et al., 2011; Niinimäki and Koivisto, 2013). However, MCMC-based methods suffer from the problem of no guarantee on the approximation quality in finite runs (the Markov chains may not mix and converge in finite runs).

Another approach proposes to construct $\mathcal{G}$ with a set of high-scoring DAGs. In particular, Tian et al. (2010) studied the idea of using the k-best DAGs for Bayesian model averaging (BMA). The estimation accuracy could be monotonically improved by spending more time to compute for larger k, and the model averaging over these k-best models achieved good accuracy in structure discovery (Tian et al., 2010). As they showed experimentally, one main advantage of constructing k-best models over sampling is that MCMC method exhibited a non-negligible variability across different runs because of the randomness nature of MCMC, while the k-best method always gave consistent estimation due to its deterministic nature.

One issue with the k-best DAG algorithm (we will call it *kBestDAG*) is that the best DAGs found actually coalesce into a fraction k of Markov equivalence classes, where the DAGs within each class represent the same set of conditional independence assertions and determine the same

statistical model. It is therefore desirable if we are able to directly find the k-best equivalence classes of Bayesian networks.

1.2.3 Scaling Up Bayesian Network Structure Learning

As mentioned, exact algorithms for Bayesian learning of structural features using DP techniques require exponential time and space. The largest problems these algorithms can solve on a typical desktop computer with a few GBs of memory do not exceed 25 variables (Koivisto, 2006a). Nowadays, real world applications easily involve thousands of variables. Thus, it is urgent to scale up the learning algorithms to meet the needs of these applications. However, there is little work done in the area of structure discovery. Instead, a lot of has been done for the model selection problem, i.e., finding the optimal Bayesian networks.

As discussed, the family of dynamic programming (DP) algorithms for optimal Bayesian network learning run in time and space of $O(n2^n)$ (Ott et al., 2004; Koivisto and Sood, 2004; Singh and Moore, 2005; Silander and Myllymäki, 2006). While both the time and space requirements grow exponentially as the number of variables n increases, it is the space requirement being the bottleneck in practice. Noting this, several techniques have been developed to reduce the space usage.

In (Malone et al., 2011), the DP algorithm for finding optimal Bayesian networks in (Singh and Moore, 2005) is improved such that only the scores and information for two adjacent layers in the recursive graph are kept in memory at once. This manipulation reduces the memory usage to $O(\binom{n}{n/2})$. And they showed the implementation of the algorithm solved a problem of 30 variables in about 22 hours using 16 GB memory. However, their implementation needs external memory (i.e., hard disk) to store the entire recursive graph. This may slow down the algorithm due to the slow access to hard disk. Further, the algorithm is not quite scalable as the $O(\binom{n}{n/2})$ space usage still grows very fast as n increases.

Alternatively, Parviainen and Koivisto (2009) proposed several schemes to trade space against time. If little space is available, a divide-and-conquer scheme recursively splits the problem to subproblems, each of which can be solved completely independently. This scheme results in time $2^{2n-s}n^{O(1)}$ in space $2^s n^{O(1)}$ for any $s = n/2, n/4, n/8, ...$, where s is the size

of the subproblems. If moderate amounts of space are available, a pairwise scheme splits the search space by fixing a class of partial orders on the set of variables. This manipulation yields run-time $2^n(3/2)^p n^{O(1)}$ in space $2^n(3/4)^p n^{O(1)}$ for any $p = 0, 1, ..., n/2$ where p is a parameter controlling the space-time trade-off. Although both schemes make it practical to solve larger problems using limited space, they make a huge sacrifice in running time.

1.2.4 Parallel Algorithms for Structure Learning and Discovery

Parallel computing aims to design systems and algorithms that use multiple processing elements simultaneously to solve a problem. It allows us to overcome the time and space limitations by using supercomputers, which are usually equipped with thousands of processors and several terabytes of memory. If the computation steps in solving a problem are independent, the running time can be significantly reduced by parallelizing the execution of these independent steps on multiple processors.

Several parallel algorithms have already been developed for solving the *structure learning* problem. First, as mentioned by the authors, the pairwise scheme proposed in (Parviainen and Koivisto, 2010) allows easy parallelization on up to 2^p processors for any $p = 0, 1, ..., n/2$. Each of the processors solves a subproblem independently in time $2^n(3/4)^p n^{O(1)}$ in space $2^n(3/4)^p n^{O(1)}$. Compared to the sequential algorithm that runs in time and space of $2^n n^{O(1)}$, the parallel efficiency is $(2/3)^p$, which is suboptimal. Further, they only implemented the pairwise scheme to compute the subproblems. Thus, although their results suggest the implementation is feasible up to around 31 variables, their estimation ignores the parallelization overhead that generally becomes problematic in parallelization. Later, Tamada et al. (2011) presented a parallel algorithm that splits the search space so that the required communication between subproblems is minimal. The overall time and space complexity is $O(n^{\sigma+1}2^n)$, where $\sigma = 0, 1..., > 0$ controls the communication-space trade-off. This algorithm, as mentioned, has slightly greater space and time complexities than the algorithm in (Parviainen and Koivisto, 2009) because of redundant calculations of DP steps. Their implementation of the algorithm was able to solve 32-node network in about 5 days 14 hours using 256 processors with 3.3 GB memory per processor. However, it did not scale well on more than 512 processors as the par-

allel efficiency decreased significantly from 0.74 on 256 processors to 0.39 on 1024 processors. Nikolova et al. (2009, 2013) described a novel parallel algorithm that realizes direct parallelization of the sequential DP algorithm in Ott et al. (2004) with optimal parallel efficiency. This algorithm is based on the observation that the subproblems constitute a lattice equivalent to an n-dimensional (n-D) hypercube, which has been proved to be a very powerful interconnection network topology used by most of modern parallel computer systems (Dally and Towles, 2004; Ananth et al., 2003; Loh et al., 2005). An advantage of this hypercube algorithm is that it does not calculate redundant steps or scores. Their implementation of the algorithm has been showed scalable on up to 2048 processors (Nikolova et al., 2013). Using 1024 processors with 512 MB memory per processor, they solved a problem with 30 variables in 1.5 hours.

In contrast, using parallel computing to speed and scale up *structure discovery* has not been studied so extensively. To our knowledge, there are no parallel algorithms developed for computing the exact posterior probability of structural features. Although Parviainen and Koivisto (2010) extended the parallelizable partial-order scheme to the *structure discovery* problem, they did not offer any explicit way to parallelize it. Although the DP algorithms for finding the optimal DAG and for the local structure discovery are analogous, they differ in some significant places. These differences prohibit the direct adaption of the existing parallel algorithms for *structure learning* to *structure discovery*.

1.3 Thesis Overview

In this dissertation, we develop algorithms to achieve more accurate, efficient and scalable structure learning and discovery in Bayesian networks. Further, we demonstrate these algorithms in applications of systems biology and educational data mining. The rest of the dissertation is organized as follows.

In chapter 2, we study the problem of learning a Bayesian network structure from the data and propose a novel heuristic algorithm that takes advantage of the idea of *curriculum learning* and learns Bayesian network structures by stages. We prove theoretical advantages of our algorithm and also empirically show that it outperforms the state-of-the-art heuristic approach in learning Bayesian network structures under several different evaluation metrics.

In chapter 3, we develop an algorithm to efficiently enumerate the k-best equivalence classes of DAGs. Our algorithm is capable of finding much more best DAGs than the previous algorithm that directly finds the k-best DAGs (Tian et al., 2010). We demonstrate our algorithm in the task of Bayesian model averaging that estimates the posterior probabilities of local structural features.

In chapter 4, we study how parallelism can be used to tackle the exponential time and space complexity in the exact Bayesian structure discovery. We present a parallel algorithm capable of computing the exact posterior probabilities of all possible directed edges with optimal parallel space efficiency and nearly optimal parallel time efficiency. We show that our algorithm can be used for discovering the *yeast* pheromone response pathways.

In chapter 5, we develop novel algorithms for exact Bayesian learning of ancestor relations in Bayesian networks. Existing algorithm assumes an order-modular prior over DAGs that does not respect Markov equivalence. Our algorithms allows uniform prior and respect the Markov equivalence. We apply our algorithm to a biology data set for discovering protein signaling pathways.

In chapter 6, we study the problem of estimating the prerequisite relationships between skills from student performance data. We introduce *Combined student Modeling and prerequisite Discovery* (COMMAND), a novel algorithm for jointly inferring a skill prerequisite graph and a student model. COMMAND learns the prerequisite relations as a Bayesian network that allows modeling of the full prerequisite structure of skills. COMMAND is useful for designing intelligent tutoring systems that assess student knowledge or that offer remediation interventions to students.

In chapter 7, we conclude the dissertation with a summary of contributions and directions for future research.

CHAPTER 2. CURRICULUM LEARNING OF BAYESIAN NETWORK STRUCTURES

In the problem of Bayesian network structure learning, one tries to find a DAG that best explains the observed data. Score-based search approach converts the learning problem to an optimization problem and attempts to maximize a scoring function over the space of all possible DAGs. Heuristic algorithms try to find a good DAG without searching the entire DAG space, thus are more efficient than exact algorithms.

In this chapter, we take advantage of the idea of *curriculum learning* and design a novel heuristic algorithm to learn a Bayesian network structure from data.

2.1 Introduction

Most of the current approaches to Bayesian network structure learning try to discover the dependency relations between *all* the variables by looking at *all* the training samples at once. This is in contrast to the way human learn. Human rarely consider all the variables as well as all the data samples at the same time; instead, they typically start with the most common subset of samples to identify the dependency relations between a small subset of variables, and only after some knowledge (i.e., a partial model) is learned would they turn to less common samples that involve additional variables. By learning in this more organized and progressive way, human are able to accumulate a large amount of knowledge both accurately and efficiently.

The learning strategy described above can be seen as an instance of curriculum learning (Bengio et al., 2009), which originates from the idea that learning starting with simpler examples or easier tasks can help obtain faster convergence and better solutions. In particular, the strategy belongs to a type of curriculum learning called incremental construction (Tu and

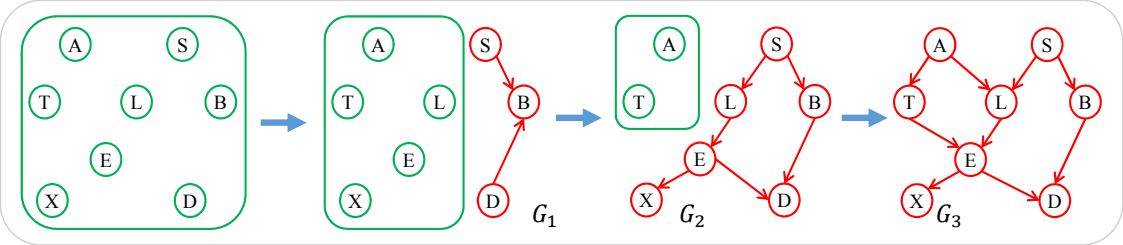


Figure 2.1: An illustrative example of curriculum learning of a Bayesian network structure. Given a curriculum $(\{S, B, D\}, \{S, B, D, L, E, X\}, \{S, B, D, L, E, X, A, T\})$, we learn the Bayesian network structure in three stages: (1) learn a subnet G_1 over $\{S, B, D\}$ from scratch; (2) learn a larger subnet G_2 over $\{S, B, D, L, E, X\}$ with G_1 as the start point of search; (3) learn a full network with G_2 as the start point. Each subnet (in red) is conditioned on the rest of the variables (in green).

Honavar, 2011), which decomposes the target structure into multiple components and learns one component at each curriculum stage.

Based on this *incremental construction* idea, we design a novel heuristic algorithm that learns the Bayesian network structure by stages. At each stage a subnet is learned over a selected subset of the random variables conditioned on fixed values of the rest of the variables. The selected subset grows with stages until it includes all the variables at the final stage. Figure 2.1 shows an illustrative example of our algorithm. We theoretically prove that the target subnet at each stage is increasingly closer to the target Bayesian network with the advance of the curriculum stages. In our experiments, we first show that not only does our algorithm learn more likely Bayesian networks given the training data, but it can also recover Bayesian networks that are better than the state-of-the-art heuristic approach with respect to both the structures of the target Bayesian networks and the distributions represented by the target Bayesian networks. We also show that our algorithm gives rise to a better classification performance when using the learned Bayesian network as classifiers.

2.2 Curriculum Learning

Humans and animals learn much better when the examples are not randomly presented but organized in a meaningful order which starts from relatively simple concepts and gradually introduces more complex ones. This idea has been formalized in the context of machine learning as *curriculum learning* (Bengio et al., 2009). A curriculum is a sequence of weighting schemes

of the training data, denoted by $(W_1, W_2, ..., W_m)$. The first scheme W_1 assigns more weight to "easier" training samples, and each next scheme assigns slightly more weight to "harder" examples, until the last scheme W_m that assigns uniform weight to all examples. How to measure the "easiness" or complexity of training samples may vary depending on the learning problems, and no general measurement has been proposed. Learning is done iteratively, each time from the training data weighted by the current weighting scheme and initialized with the learning result from the previous iteration.

Curriculum learning has been successfully applied to many problems, such as learning language models and grammars (Elman, 1993; Spitkovsky et al., 2010; Tu and Honavar, 2011) and object recognition and localization (Kumar et al., 2010). There have also been attempts to explain the advantages of curriculum learning. Bengio et al. (2009) proposed that a well chosen curriculum strategy can act as a continuation method (Allgower and Georg, 1990), which first optimizes a highly smoothed objective and then gradually considers less smoothing. Tu and Honavar (2011) contended that in learning structures such as grammars, an ideal curriculum decomposes the structure into multiple components and guides the learner to incrementally construct the target structure. More recently, a few extensions of the original idea of curriculum learning have been proposed (Kumar et al., 2010; Jiang et al., 2015).

2.3 Curriculum Learning of Bayesian Network Structures

The basic idea to apply curriculum learning and incremental construction in Bayesian network structure learning is that we can define a sequence of intermediate learning targets $(G_1, ..., G_m)$, where each G_i is a subnet of the target Bayesian network over a subset of variables $\mathbf{X}_{(i)}$ conditioned on certain fixed values $\mathbf{x}'_{(i)}$ of the rest of the variables $\mathbf{X}'_{(i)}$, where $\mathbf{X}_{(i)} \subseteq \mathbf{X}$, $\mathbf{X}'_{(i)} = \mathbf{X} \setminus \mathbf{X}_{(i)}$ and $\mathbf{X}_{(i)} \subset \mathbf{X}_{(i+1)}$; at stage i of curriculum learning, we try to learn G_i from a subset of data samples with $\mathbf{X}'_{(i)} = \mathbf{x}'_{(i)}$. In terms of the sample weighting scheme $(W_1, W_2, ..., W_m)$, each W_i assigns 1 to those samples with $\mathbf{X}'_{(i)} = \mathbf{x}'_{(i)}$ and 0 to the other samples.

However, training samples are often very limited in practice and thus the subset of samples with $\mathbf{X}'_{(i)} = \mathbf{x}'_{(i)}$ would typically be very small. Learning from such small-sized training sample is deemed unreliable. A key observation is that when we fix $\mathbf{X}'_{(i)}$ to different values, our learning target is actually the same DAG structure G_i but with different parameters (CPDs). Thus, we can make use of all the training samples in learning G_i at each stage by revising the scoring function to take into account multiple versions of parameters. This strategy extends the original curriculum learning framework.

Note that in the ideal case, the subnet learned in each stage would have only one type of discrepancy from the truth Bayesian network: it would contain extra edges between variables in $\mathbf{X}_{(i)}$ due to conditioning on $\mathbf{X}'_{(i)}$. More specifically, such variables in $\mathbf{X}_{(i)}$ must share a child node that is in, or has a descendant in, $\mathbf{X}'_{(i)}$ such that they are not d-separated when conditioning on $\mathbf{X}'_{(i)}$.

2.3.1 Scoring Function

In this paper, we use Bayesian score to design a scoring function that uses all training samples. Assume the domain for $\mathbf{X}'_{(i)}$ is $\{\mathbf{x}'_{(i),1}, ..., \mathbf{x}'_{(i),q}\}$. Then we can have a set of data segments $D_i = \{D_{i,1}, ..., D_{i,q}\}$ by grouping samples based on the values of $\mathbf{X}'_{(i)}$ and then projecting on $\mathbf{X}_{(i)}$. Assuming $D_{i,1}, ..., D_{i,q}$ are generated by the same DAG G_i but with "independent" CPDs, we can derive

$$P(G_i, D_i) = P(G_i) \prod_{j=1}^{q} P(D_{i,j}|G_i) = P(G_i)^{1-q} \prod_{j=1}^{q} P(G_i, D_{i,j}). \tag{2.1}$$

If we take logarithm for both sides, we obtain

$$\log P(G_i, D_i) = (1-q) \log P(G_i) + \sum_{j=1}^{q} \log P(G_i, D_{i,j}). \tag{2.2}$$

If we set uniform prior for G_i, i.e., $P(G_i) \propto 1$, we then have

$$\log P(G_i, D_i) = C + \sum_{j=1}^{q} \log P(G_i, D_{i,j}), \tag{2.3}$$

where C is a constant. We use BDeu score (Buntine, 1991) for discrete variables, i.e., $\log P(G_i, D_{i,j}) = score_{BDe}(G_i, D_{i,j})$, so we have the following scoring function:

$$score(G_i, D_i) = \sum_{j=1}^{q} score_{BDe}(G_i, D_{i,j}), \tag{2.4}$$

i.e., the sum of BDe scores which are individually evaluated on each of the data segments $D_{i,1}, ..., D_{i,q}$.

One common problem with curriculum learning is that the learner may overfit the intermediate learning targets, especially when the number of variables is large and thus we have to divide learning into many stages. Overfitting also occurs when the sample size is small. Therefore, we introduce a penalty function that penalizes the size of the network especially when the number of variables is large or the sample size is small

$$Penalty(G_i : D_i) = \left(\frac{a}{SS} + \frac{V(G_i)}{b}\right) E(G_i), \tag{2.5}$$

where SS is the sample size, $V(G_i)$ and $E(G_i)$ denote the number of variables and number of edges in G_i respectively, and a and b are positive constants. The penalty function is proportional to $E(G_i)$ which represents the complexity of the network. The impact of $E(G_i)$ is enlarged with either larger number of variables or smaller sample size. Combined with the penalty function, the scoring function becomes

$$score(G_i : D_i) = \sum_{j=1}^{q} score_{BDe}(G_i, D_{i,j}) - \left(\frac{a}{SS} + \frac{V(G_i)}{b}\right) E(G_i). \tag{2.6}$$

2.3.2 Curriculum

A remaining fundamental question is: what *curriculum*, i.e., the sequence of variable sets $(\mathbf{X}_{(1)}, ..., \mathbf{X}_{(m)})$, shall we use? Or equivalently, from stage $i-1$ to i, which variables $\mathbf{X}_{(i-1,i)}$ should we select to produce $\mathbf{X}_{(i)} = \mathbf{X}_{(i-1)} \cup \mathbf{X}_{(i-1,i)}$?

Intuitively, we should select the variables that are most connected to the current set of variables $\mathbf{X}_{(i-1)}$, because otherwise we may learn more edges that do not exist in the target Bayesian network. Without knowing the true connectivity, we can measure the strength of the dependency (e.g., using mutual information) with the current set of variables to heuristically

estimate the connectivity. Given a set of variables $\mathbf{X}$, for every variable $Y \in \mathbf{X} \setminus \mathbf{X}_{(i-1)}$, we define the average pairwise mutual information by

$$AveMI(Y, \mathbf{X}_{(i-1)}) = \sum_{X \in \mathbf{X}_{(i-1)}} MI(X, Y)/|\mathbf{X}_{(i-1)}|. \tag{2.7}$$

In stage i, we select $\mathbf{X}_{(i-1,i)}$ in the following way. We first pick variable Y_1 with the largest $AveMI(Y_1, \mathbf{X}_{(i-1)})$; we then pick the second variable Y_2 with the largest $AveMI(Y_2, \mathbf{X}_{(i-1)} \cup \{Y_1\})$; this is repeated until we have picked a pre-specified number of variables. The number of variables selected, $|\mathbf{X}_{(i-1,i)}|$, is called the *step size* and is a parameter of our algorithm. The step size can be a constant, meaning that we add the same number of variables in each stage. Or it can be different among stages. Intuitively, the smaller the step size is, the more cautious and less time-efficient the algorithm is, and also the more likely the algorithm would overfit the intermediate Bayesian networks.

Note that in the first stage, $\mathbf{X}_{(i-1)} = \mathbf{X}_{(0)}$ is an empty set and thus we cannot select the first variable $Y_1 \in \mathbf{X} \setminus \mathbf{X}_{(0)}$ by computing $AveMI(Y_1, \mathbf{X}_{(0)})$. Instead, we select the variable with the largest $AveMI$ with all the other variables in $\mathbf{X}$ and then select additional variables in the sequential way as described above. The size s of subset $\mathbf{X}_{(1)}$ determines how large the initial subnet we start with. When $s = |\mathbf{X}|$, the algorithm learns the network in one step. When $s = 2$, the subnet has only two variables and learning is trivial. Thus, in our design we set $s = 3$.

The details of constructing a curriculum is provided in Algorithm 2.1. We now provide an example of making a curriculum. Assume $\mathbf{X} = \{A, S, T, L, B, E, X, D\}$. S has the largest $AveMI$ among all the variables, so we get the initial sequence $\mathbf{I} = (S)$. Then we calculate $AveMI$ for every variable in $\{A, T, L, B, E, X, D\}$ with $\mathbf{I}$, and find that B has the largest $AveMI$. So we append B to $\mathbf{I}$. Repeating the procedure described above, we finally get $\mathbf{I} = (S, B, D, L, E, X, A, T)$, in which every variable is the one that is most likely to have connections with the variables before it. If we initialize $\mathbf{X}_{(1)}$ with 3 variables ($\mathbf{X}_{(1)} = \{S, B, D\}$) and set the step size to 2, our curriculum would be $(\{S, B, D\}, \{S, B, D, L, E\}, \{S, B, D, L, E, X, A\}, \{S, B, D, L, E, X, A, T\})$.

Algorithm 2.1 Construct a curriculum: $ConstructCurriculum(\mathbf{X}, D, s, t)$

Input: variable set $\mathbf{X}$, training data D, size of $\mathbf{X}_{(1)}$ s, step size t.
for $i, j \in \{1, ..., n\}$ *and* $i \neq j$ **do**
    Compute empirical mutual information $MI(X_i, X_j)$
end for
$X^* \leftarrow \text{argmax}_X \, AveMI(X, \mathbf{X} \setminus X)$
$\mathbf{X}_{(1)} \leftarrow \{X^*\}$
$\mathbf{Y} \leftarrow \mathbf{X} \setminus \mathbf{X}_{(1)}$
$i \leftarrow 1$
for $i < s$ **do**
    $Y^* \leftarrow \text{argmax}_{Y \in \mathbf{Y}} \, AveMI(Y, \mathbf{X}_{(1)})$
    $\mathbf{X}_{(1)} = \mathbf{X}_{(1)} \cup \{Y^*\}$
    $\mathbf{Y} \leftarrow \mathbf{Y} \setminus \{Y^*\}$
    $i \leftarrow i + 1$
end for
$m \leftarrow \lceil \frac{(n-s)}{t} + 1 \rceil$
$i \leftarrow 2$
for $i \leq m$ **do**
    $\mathbf{X}_{(i)} = \mathbf{X}_{(i-1)}$
    $j \leftarrow 0$
    for $j < t$ **do**
        if $\mathbf{Y} \neq \emptyset$ **then**
            $Y^* \leftarrow \text{argmax}_{Y \in \mathbf{Y}} \, AveMI(Y, \mathbf{X}_{(i)})$
            $\mathbf{X}_{(i)} = \mathbf{X}_{(i)} \cup \{Y^*\}$
            $\mathbf{Y} \leftarrow \mathbf{Y} \setminus \{Y^*\}$
            $j \leftarrow j + 1$
        end if
    end for
    $i \leftarrow i + 1$
end for
return $(\mathbf{X}_{(1)}, ..., \mathbf{X}_{(m)})$

2.3.3 Algorithm

Given the training data D, size of initial set $\mathbf{X}_{(1)}$ and step size t, we first construct the curriculum $(\mathbf{X}_{(1)}, ..., \mathbf{X}_{(m)})$, where $\mathbf{X}_{(m)} = \mathbf{X}$. Then we run the *MMPC* algorithm in (Tsamardinos et al., 2003) to generate the parents and children (PC) set S_i for each X_i. Let $\mathbf{S} = (S_1, ..., S_n)$. In each learning stage of the curriculum, we use score-based search to find a good partial network with the partial network learned in the previous stage plus the new variables with no edge attached as the start point. Algorithm 2.2 sketches our algorithm, in which $search(D_i, \mathbf{X}_{(i)}, \mathbf{S}, G_{i-1})$ can be any search algorithm that starts from G_{i-1} and searches the space of DAGs over variables $\mathbf{X}_{(i)}$ to optimize our scoring function with training data D_i. We use PC set $\mathbf{S}$ to constrain the search, i.e., for variable X_i, only edges included in its PC set S_i are considered during the search. This helps prevent adding too many spurious edges. In our implementation, we simply use greedy hill climbing as the search algorithm.

In some stages, the number of data segments does not change although additional variables are selected. In this case, it can be shown that the subnet learned in the previous stage plus the new variables with no edge attached is already a local optimum, and therefore we can skip the stage without changing the learning result.

Algorithm 2.2 Curriculum Learning of Bayesian network structure

1: **Input:** variable set $\mathbf{X}$, training data D, size of $\mathbf{X}_{(1)}$ s, step size t.
2: $(\mathbf{X}_{(1)}, ..., \mathbf{X}_{(m)}) \leftarrow ConstructCurriculum(\mathbf{X}, D, s, t)$
3: **for** $i \in \{1, ..., n\}$ **do**
4: $\quad S_i \leftarrow MMPC(X_i, D)$
5: **end for**
6: $\mathbf{S} \leftarrow (S_1, ..., S_n)$
7: Initialize G_0 to a network containing variables in $\mathbf{X}_{(1)}$ with no edge.
8: $i \leftarrow 1$
9: **for** $i \leq m$ **do**
10: $\quad$ Generate the set of data segments $D_i = \{D_{i,1}, ..., D_{i,q}\}$ based on the values of $\mathbf{X} \setminus \mathbf{X}_{(i)}$
11: $\quad G_i \leftarrow search(D_i, \mathbf{X}_{(i)}, \mathbf{S}, G_{i-1})$
12: $\quad i \leftarrow i + 1$
13: **end for**
14: **return** G_m

2.4 Theoretical Analysis

Curriculum learning specifies a sequence of intermediate learning targets. Ideally, each intermediate target should be closer to the subsequent targets than any of its predecessors in the sequence. In this section we show that our curriculum learning approach to learning Bayesian networks satisfies this desired property.

With Bayesian networks as our learning targets, there are two different ways to measure the distance between them. The first is to measure the distance between the structures of two Bayesian networks. One such distance measure is the structural Hamming distance (*SHD*) (Tsamardinos et al., 2006), which measures the number of extra, missing or differently oriented edges between the two CPDAGs that respectively represent the equivalence classes of two Bayesian networks. The second is to measure the distance between the probabilistic distributions defined by two Bayesian networks. One such distance measure is the total variation distance (Csisz et al., 1967). With discrete random variables, the total variation distance between two distributions can be defined as:

$$d_{TV}(P, Q) = \frac{1}{2} \sum_{\mathbf{X}} |P(\mathbf{X}) - Q(\mathbf{X})| .$$

Below we analyze our curriculum learning approach based on these two types of distance measures respectively and show that our approach satisfies the desired property based on both distance measures.

2.4.1 Analysis Based on Distance between Structures

Suppose $\mathbf{X}_{(i)}$ is the set of variables selected in curriculum stage i and $\mathbf{X}'_{(i)} = \mathbf{X} \setminus \mathbf{X}_{(i)}$ is the rest of the variables. Recall that we try to learn a subnet of the true Bayesian network over variables in $\mathbf{X}_{(i)}$ that is *conditioned on* fixed values of variables in $\mathbf{X}'_{(i)}$. Therefore, the actual learning target at stage i is a Bayesian network G_i such that: (a) between variables in $\mathbf{X}_{(i)}$, the edges are connected in accordance with the true Bayesian network except that there might be extra edges between variables that share one or more descendants in $\mathbf{X}'_{(i)}$ (recall that the values of the variables in $\mathbf{X}'_{(i)}$ are fixed at stage i); (b) the variables in $\mathbf{X}'_{(i)}$ are fully connected with each other (because at stage i we regard the joint assignments to the variables in $\mathbf{X}'_{(i)}$ as

the conditions and do not model any conditional independence between them); (c) there is an edge between each variable in $\mathbf{X}_{(i)}$ and each variable in $\mathbf{X}'_{(i)}$ (because the subnet over $\mathbf{X}_{(i)}$ is conditioned on all the variables in $\mathbf{X}'_{(i)}$). The orientation of the edges described in (b) and (c) can be arbitrary since it is not actually to be learned at stage i , but if we assume that these edges are oriented in a way that is consistent with the true Bayesian network then we have the following theorem.

Theorem 2.1. *For any i, j, k s.t. $1 \leq i < j < k \leq n$, we have*

$$d_H(G_i, G_k) \geq d_H(G_j, G_k)$$

where $d_H(G_i, G_j)$ is the structural Hamming distance (SHD) between the structures of two Bayesian networks G_i and G_j.

Proof. At each stage of the curriculum, a set of variables $\mathbf{V} = \mathbf{X}_{(i)} \setminus \mathbf{X}_{(i-1)}$ become selected. This leads to two changes to the intermediate target Bayesian network: first, some extra edges between variables in $\mathbf{X}_{(i-1)}$ that share descendants in $\mathbf{V}$ are removed because their descendants no longer have fixed values; second, some edges connected to variables in $\mathbf{V}$ are removed to make the subnet of the variables in $\mathbf{X}_{(i)}$ consistent with the true Bayesian network. In other words, we always remove edges and never add or re-orient any edge of the Bayesian network at each stage of the curriculum. Since the corresponding CPDAG has the same structure as the Bayesian network except for some edges becoming undirected, it can also be shown that only edge-removal occurs to the CPDAG at each stage of the curriculum. Therefore, the structural Hamming distance $d_H(G_i, G_j)$ is simply the number of edges removed during stages $i+1$ to j. Since $i < j < k$, the set of edges removed during stages $i+1$ to k is a superset of the set of edges removed during stages $j+1$ to k. Therefore, we have $d_H(G_i, G_k) \geq d_H(G_j, G_k)$. □

2.4.2 Analysis Based on Distance between Distributions

Based on the discussion in the previous subsection, it can be seen that the intermediate learning target G_i of stage i represents a probabilistic distribution $P(\mathbf{X}_{(i)}|\mathbf{X}'_{(i)})Q(\mathbf{X}'_{(i)})$, where P denotes the true conditional distribution of $\mathbf{X}_{(i)}$ given $\mathbf{X}'_{(i)}$ as represented by the target

Bayesian network, and Q denotes an estimated distribution over $\mathbf{X}'_{(i)}$ (e.g., simply estimated based on the histogram built from the training data). We can prove the following theorem.

Theorem 2.2. *For any i, j, k s.t. $1 \leq i < j < k \leq n$, we have*

$$d_{TV}(G_i, G_k) \geq d_{TV}(G_j, G_k)$$

where $d_{TV}(G_i, G_j)$ is the total variation distance between the two distributions defined by the two Bayesian networks G_i and G_j.

Proof. For any $i < j$, let $\mathbf{Y}_{ij} = \mathbf{X}_{(j)} \setminus \mathbf{X}_{(i)}$. We have

$$\begin{aligned} d_{TV}(G_i, G_j) &= \frac{1}{2} \sum_{\mathbf{X}} \left| P(\mathbf{X}_{(i)} | \mathbf{X}'_{(i)}) Q(\mathbf{X}'_{(i)}) - P(\mathbf{X}_{(j)} | \mathbf{X}'_{(j)}) Q(\mathbf{X}'_{(j)}) \right| \\ &= \frac{1}{2} \sum_{\mathbf{X}} P(\mathbf{X}_{(i)} | \mathbf{X}'_{(i)}) \left| Q(\mathbf{X}'_{(i)}) - P(\mathbf{Y}_{ij} | \mathbf{X}'_{(j)}) Q(\mathbf{X}'_{(j)}) \right| \\ &= \frac{1}{2} \sum_{\mathbf{X}'_{(i)}} \left| Q(\mathbf{X}'_{(i)}) - P(\mathbf{Y}_{ij} | \mathbf{X}'_{(j)}) Q(\mathbf{X}'_{(j)}) \right| \end{aligned}$$

Therefore, we have

$$d_{TV}(G_i, G_k) = \frac{1}{2} \sum_{\mathbf{X}'_{(j)}} \sum_{\mathbf{Y}_{ij}} \left| Q(\mathbf{X}'_{(i)}) - P(\mathbf{Y}_{ik} | \mathbf{X}'_{(k)}) Q(\mathbf{X}'_{(k)}) \right|$$

and

$$\begin{aligned} d_{TV}(G_j, G_k) &= \frac{1}{2} \sum_{\mathbf{X}'_{(j)}} \left| Q(\mathbf{X}'_{(j)}) - P(\mathbf{Y}_{jk} | \mathbf{X}'_{(k)}) Q(\mathbf{X}'_{(k)}) \right| \\ &= \frac{1}{2} \sum_{\mathbf{X}'_{(j)}} \left| \sum_{\mathbf{Y}_{ij}} Q(\mathbf{X}'_{(i)}) - \sum_{\mathbf{Y}_{ij}} P(\mathbf{Y}_{ik} | \mathbf{X}'_{(k)}) Q(\mathbf{X}'_{(k)}) \right| \end{aligned}$$

Because the absolute value is sub-additive, we have

$$\sum_{\mathbf{Y}_{ij}} \left| Q(\mathbf{X}'_{(i)}) - P(\mathbf{Y}_{ik} | \mathbf{X}'_{(k)}) Q(\mathbf{X}'_{(k)}) \right| \geq \left| \sum_{\mathbf{Y}_{ij}} \left(Q(\mathbf{X}'_{(i)}) - P(\mathbf{Y}_{ik} | \mathbf{X}'_{(k)}) Q(\mathbf{X}'_{(k)}) \right) \right|$$

Therefore,

$$d_{TV}(G_i, G_k) \geq d_{TV}(G_j, G_k)$$

□

2.5 Experiments

In this section, we empirically evaluate our algorithm and compare it with MMHC (Tsamardinos et al., 2006), the current state-of-the-art heuristic algorithm in Bayesian network structure learning. For both algorithms, we used BDeu score as given in Equation 1.4 with the equivalent sample size 10 as the scoring function and used the MMPC module included in Causal Explorer (Aliferis et al., 2003) with the default setting to generate the PC set. For MMHC, we used the settings mentioned by Tsamardinos et al. (2006).

We conducted two sets of experiments that evaluated the learned Bayesian networks in different ways. In the first set of experiments we learned from synthetic training data sampled from ground-truth Bayesian networks and then compared the network structures recovered by our curriculum-based learning algorithm (CL) and MMHC using a set of standard Bayesian network structure evaluation metrics. In the second set of experiments, we learned from the real data for classification tasks, used the learned Bayesian network structures as classifiers and then compared the classification performance.

When running our algorithm on the datasets, we set the step size (introduced in section 2.3.2) to 1, 2 and 3 and learned three Bayesian networks; we also learned a Bayesian network by hill climbing with no curriculum. We then picked the Bayesian network with the largest BDeu score as the final output. We tuned the parameter a and b of the penalty function (Equation 2.5) on a separate validation set and fixed them to 1000 and 100 respectively.

2.5.1 Experiments on Bayesian Network Reconstruction

The ability of Bayesian network structure learning algorithms to recover Bayesian network structures from training data randomly sampled from the ground-truth networks with known structures and parameters could be used to measure the quality of the learning algorithms.

2.5.1.1 Experimental Setup

We collected 10 benchmark Bayesian networks from the `bnlearn` repository[1]. The statistics of these Bayesian networks are shown in Table 2.1. From each of these Bayesian networks, we generated datasets of various sample sizes ($SS = 100, 500, 1000, 5000, 10000, 50000$). For each sample size, we randomly generated 5 datasets and reported the algorithm performance averaged over these 5 datasets.

Table 2.1: Bayesian networks used in experiments.

Network	Num. vars	Num. edges	Max in/out-degree	Cardinality range	Average cardinality
alarm	37	46	4/5	2-4	2.84
andes	223	338	6/12	2-2	2.00
asia	8	8	2/2	2-2	2.00
child	20	25	2/7	2-6	3.00
hailfinder	56	66	4/16	2-11	3.98
hepar2	70	123	6/17	2-4	2.31
insurance	27	52	3/7	2-5	3.30
sachs	11	17	3/6	3-3	3.00
water	32	66	5/3	3-4	3.63
win95pts	76	112	7/10	2-2	2.00

Cardinality denotes the number of values that a variable can take.

2.5.1.2 Evaluation Metrics

We used four metrics to evaluate the learned Bayesian networks: BDeu, BIC, KL and SHD. The first three metrics were evaluated on a separate test dataset of 5000 samples for each Bayesian network. The BDeu score, the scoring function used in our learning algorithms, measures how likely the network is given the data. BIC (Bayesian information criterion) can be regarded as the likelihood of the learned structure after having seen the data with a penalty term of model complexity measured by the number of parameters:

$$BIC(G:D) = \sum_{i=1}^{n}\sum_{j=1}^{q_i}\sum_{k=1}^{r_i} N_{ijk}\log\frac{N_{ijk}}{N_{ij}} - \frac{1}{2}\log(N)\sum_{i=1}^{n}(r_i-1)q_i, \tag{2.8}$$

[1] http://www.bnlearn.com/bnrepository/.

where n denotes the number of variables, N is the number of samples in the test dataset, r_i denotes the number of values that X_i can take, $q_i = \prod_{X_l \in Pa_i} r_l$ denotes the number of values that the parent set Pa_i of X_i can take, N_{ijk} is the number of samples in D where $X_i = k$ and $Pa_i = j$, and N_{ij} is the number of samples with $Pa_i = j$ in D.

Both BDeu and BIC have the limitation that they are only reasonable under certain assumptions. To directly measure how close the gold-standard network and the learned network are, we used Kullback-Leibler divergence (KL) between the joint probability distributions associated respectively with the true network(P_T) and the learned network(P_L):

$$KL(P_T, P_L) = \sum_{\mathbf{X}} P_T(\mathbf{X}) \log \left(\frac{P_T(\mathbf{X})}{P_L(\mathbf{X})} \right). \tag{2.9}$$

For the convenience of estimation, we used an equivalent form of Equation 2.9 by Acid and de Campos (2001):

$$KL(P_T, P_L) = -H_{P_T}(\mathbf{X}) + \sum_{X_i \in \mathbf{X}} H_{P_T}(X_i) - \sum_{X_i \in \mathbf{X}, Pa_L(X_i) \neq \emptyset} MI_{P_T}(X_i, Pa_L(X_i)), \tag{2.10}$$

where H_{P_T} denotes the Shannon entropy with respect to P_T. In Equation 2.10, the first two terms are not dependent on the learned network, so following Tsamardinos et al. (2006), we only calculate and report the last term of the equation. Note that the last term appears with a negative sign, and hence the higher its value is, the smaller the KL-divergence is and the closer the learned network is to the true network.

Structural Hamming distance (SHD) is another distance metric, which directly measures the difference between the structures of the two networks as explained in Section 2.4.

2.5.1.3 Results

Table 2.2 shows the comparison between our algorithm (CL) and MMHC. Note that we choose to show the average ratios between the raw scores and the corresponding scores of CL. This is because the raw scores from different datasets vary significantly in order of magnitude, and the average of raw scores would be dominated by those from a small subset of the datasets. It can be seen that CL outperforms MMHC in almost all the cases, in terms of both the scores and the number of winning networks. A notable exception is that when $SS = 100$, CL under-performs MMHC on all the networks for three of the four metrics.

Table 2.2: Comparison between CL and MMHC on four metrics

Metric	Algorithm	Sample Size (SS) 100	500	1000	5000	10000	50000
BDeu	CL	1(0)	1(10)	1(9)	1(8)	1(9)	1(7)
	MMHC	0.89(10)	1.06(0)	1.02(1)	1.01(2)	1.01(1)	1.01(3)
BIC	CL	1(0)	1(9)	1(9)	1(6)	1(7)	1(7)
	MMHC	0.88(10)	1.07(1)	1.02(1)	1.02(4)	1.01(3)	1.01(3)
KL	CL	1(0)	1(10)	1(9)	1(7)	1(9)	1(9)
	MMHC	1.72(10)	0.82(0)	0.96(1)	0.96(2)	0.97(0)	0.96(0)
SHD	CL	1(7)	1(9)	1(7)	1(7)	1(8)	1(6)
	MMHC	1.06(3)	1.26(1)	1.29(3)	1.07(2)	1.22(1)	1.24(3)

Each number is an average normalized scores, i.e., the average of the ratios between the raw scores and the corresponding scores of CL (the ratios are averaged over 10 networks and 5 runs with randomly sampled training datasets on each network). For BDeu, BIC and SHD, smaller ratios indicate better learning results; for KL, larger numbers indicate better learning results. Each number in parentheses indicates the number of winning networks among the 10 networks, i.e., on how many networks the algorithm produced better results than its competitor. The number of draws (networks with equal scores) are not counted.

We find that it is mainly because the penalty term (Equation 2.5) becomes too large when SS is very small, which drives the learner to produce a network with few edges. For example, on the Andes network with $SS = 100$, the learned network contains only around 50 edges while the number of edges in the true network is 338.

Since SHD is one of the most widely used evaluation metrics for Bayesian network structure learning, we further investigate the SHD scores of the two algorithms under different settings. Figure 2.2 plots the SHD averaged over five runs on the Andes, Hailfinder, Hepar2 and Win95pts networks. It again shows that CL outperforms MMHC in almost all the cases.

With respect to running-time, our algorithm is in general slower than MMHC, on average taking 2.7 times as much time. One reason is that our algorithm has to perform hill climbing for multiple times, once at each stage, and the number of stages is proportional to the number of variables. Another reason is that our scoring function takes more time to compute: we have to compute a separate score for each data segment, which becomes slow when the data is partitioned into too many segments. The number of segments is determined by the number

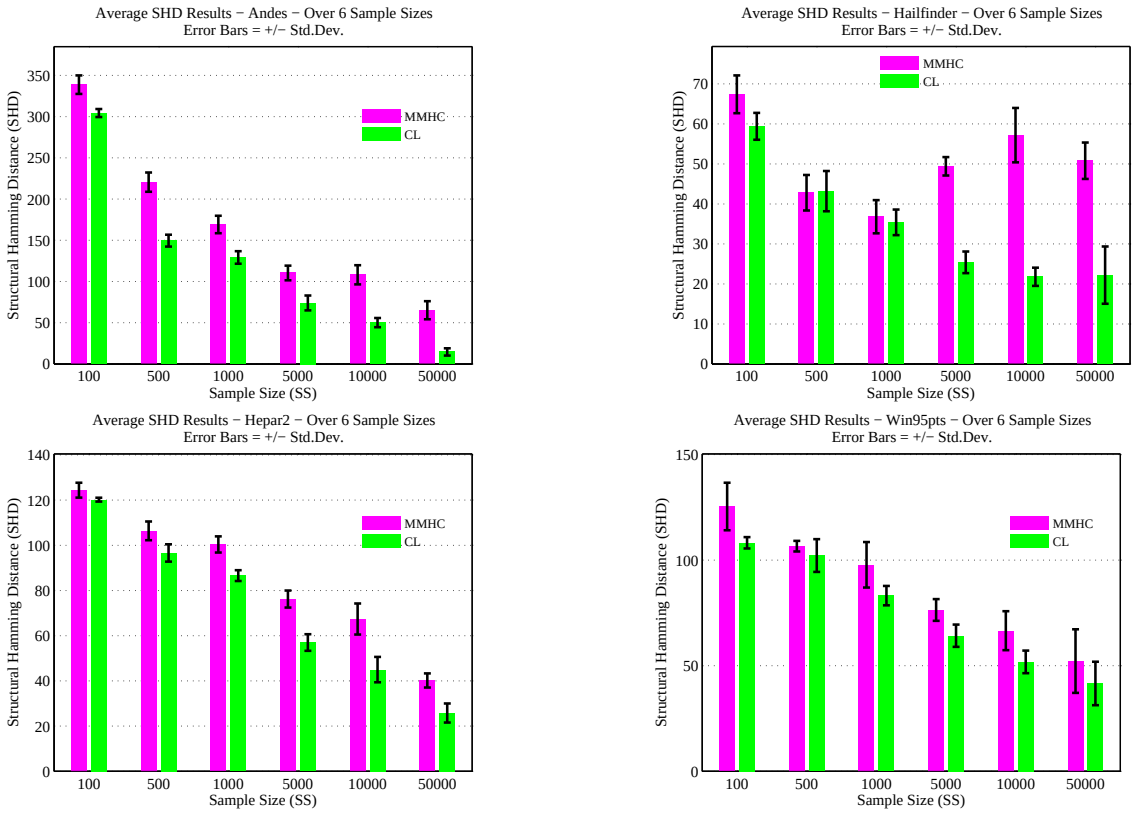


Figure 2.2: Comparison of the average SHD on the Andes, Hailfinder, Hepar2 and between CL and MMHC.

of variables as well as the cardinality of each variable. Our experiments show that the average cardinality of variables has a larger impact to the running time of our algorithm than the number of variables. With $SS = 5000$, the Andes network (223 variables with average cardinality of 2) takes only a few minutes for our algorithm to run, while the Mildew network (35 variables with average cardinality of 17.6) takes a few hours. To verify that the good performance of our algorithm does not come from the extra running time, we ran TABU search[2] of Bayesian network structures on each dataset with the same amount of time as used by our algorithm and found that our algorithm still has significantly better performance.

[2]TABU search augments greedy hill-climbing by allowing worsening moves and using a *tabu* list to keep track of and avoid recently visited solutions.

2.5.1.4 Analysis of Step Size

The *step size* defined in Section 2.3.2 is the number of variables added in each stage. As described earlier, we ran CL with different step sizes as well as hill-climbing without curriculum and picked the final Bayesian network with the best BDeu Score. Here we analyze the frequency of each step size leading to the best score and show results.

Table 2.3 shows detailed statistics of the winning step size, which is the step size that gives rise to the best score. It can be seen that only on a small fraction (54 out of 300) of the datasets did hill climbing with no curriculum produce the best score, implying that CL indeed helps to find the better Bayesian network structures.

Table 2.3: Frequency of the winning step size

	Step Size (t)			
Network	No CL	1	2	3
alarm	1	18	9	2
asia	5	21	2	2
insurance	8	7	11	4
child	3	17	6	4
sachs	8	12	6	4
water	7	11	7	5
hepar2	5	8	7	10
win95pts	1	15	5	9
hailfinder	16	14	0	0
andes	0	13	10	7
total	54	136	63	47
ratio	18.00%	45.00%	21.00%	16.00%

Each row shows, for a given Bayesian network, the number of times the corresponding step size produces the best score. *Ratio* is the percentage of the $10 \times 5 \times 6 = 300$ datasets (number of the Bayesian networks times number of datasets times number of the sample sizes) on which CL of a specific step size (or without curriculum, No CL) produces the best score.

2.5.1.5 Theory Verification

In section 2.4 we have proved that each intermediate target Bayesian network in our curriculum is closer to the subsequent target Bayesian networks than any of its predecessors. Here we would like to empirically demonstrate that the learner is indeed guided by these intermediate target Bayesian networks to produce intermediate learning results that become increasingly

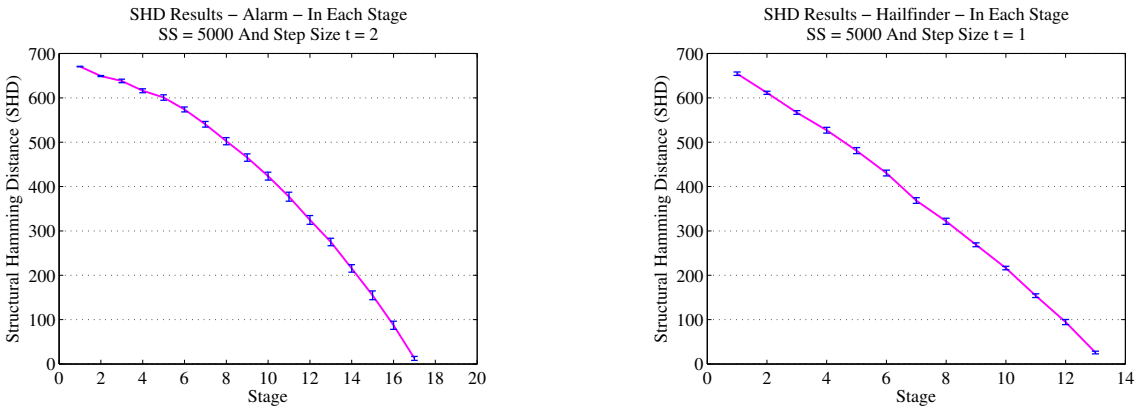


Figure 2.3: Changes of SHD from the target Bayesian network during curriculum learning with $SS = 5000$ on the Alarm and Hailfinder networks.

closer to the target Bayesian network with more curriculum stages. Note that while at stage i of the curriculum we learn a subnet over the selected variables $\mathbf{X}_{(i)}$, this subnet is conditioned on fixed values of the rest of the variables $\mathbf{X}'_{(i)} = \mathbf{X} \setminus \mathbf{X}_{(i)}$. Hence we can view the intermediate learning result at stage i as a Bayesian network over all the variables consisting of three parts: (a) the learned subnet over $\mathbf{X}_{(i)}$; (b) a fully connected subnet over $\mathbf{X}'_{(i)}$; (c) a fully connected bipartite network between $\mathbf{X}_{(i)}$ and $\mathbf{X}'_{(i)}$. In order to correctly measure the distances between the intermediate learning results and the target Bayesian network, we first randomly generated a fully connected Bayesian network over all the variables, and then at each stage i we replaced the local structure over $\mathbf{X}_{(i)}$ with the subnet that we have learned and adjusted the direction of the edges between $\mathbf{X}_{(i)}$ and $\mathbf{X}'_{(i)}$ to guarantee no directed cycles would be introduced. Figure 2.3 plots the SHD between the intermediate learning result at each stage and the target Bayesian network on two different networks. It can be seen that the intermediate learning results indeed become closer to the learning target with more curriculum stages.

2.5.2 Experiments on Classification

To further explore the advantages of CL, we evaluated it by its performance in real applications of Bayesian networks. Specifically, since Bayesian networks can be used as a classifier, we compared the classification performance of the Bayesian networks learned from CL and MMHC.

Letting $c^\star$ denote the predictive value of the class variable C given the instance $\mathbf{x}$ of $\mathbf{X}$, classification using Bayesian network in our experiments is given by

$$c^\star = \arg\max_c P(C = c|\mathbf{x}) = \arg\max_c P(C = c, \mathbf{x}), \tag{2.11}$$

where Bayesian network is learned over the variable set $\mathbf{X} \cup C$.

2.5.2.1 Experimental Setup

We performed classification experiments on 28 datasets which were obtained from the `KEEL-dataset` repository (Alcalá et al.,). The statistics of these datasets are shown in Table 2.4. We randomly sampled three fifths of the dataset as the training set and used the rest as the test set. Since our algorithm currently does not deal with continuous variables, we used the discretization method proposed by (Irani, 1993) and its implementation in `Weka` (Hall et al., 2009) to discretize continuous variables. In learning Bayesian network structures, we kept the settings of CL and MMHC same as that described at the beginning of Section 2.5 and used add-1 smoothing for unseen patterns when computing CPDs.

Two metrics are used to measure the classification accuracy: predictive accuracy (ACC) and conditional log likelihood (CLL) of the correct class given $\mathbf{x}$. Note that there are cases in which ACCs given by two different Bayesian networks are the same. Therefore, we introduce CLL that measures the confidence level of the correct class, which is more sensitive to the difference in the Bayesian networks. Given the test set D_T, we define

$$CLL(D_T) = \frac{1}{|D_T|} \sum_{\mathbf{x} \in D_T} \log P(C_\mathbf{x}|\mathbf{x}), \tag{2.12}$$

where $C_\mathbf{x}$ is the class to which $\mathbf{x}$ corresponds.

We also included the Naive Bayesian (NB) classifier as the baseline. NB can be seen as the special case of Bayesian network which has a fixed structure that takes the class variable C as the parent of all the other variables $\mathbf{X}$ with no additional edges between $\mathbf{X}$. We take the ratios between the raw scores of CL and MMHC and the corresponding scores of NB averaged over 28 datasets as the reported results.

Table 2.4: Datasets used in classification experiments.

Dataset	Num. vars	Num. classes	Num.training samples
abalone	8	28	2504
adult	14	2	27133
banana	2	2	3180
chess	36	2	1917
connect-4	42	3	40534
fars	29	8	60580
kddcup	41	23	296412
kr-vs-k	6	18	16833
magic	10	2	11412
titanic	3	2	1320
mushroom	22	2	3386
nursery	8	5	7776
phoneme	5	2	3242
ring	20	2	4440
segment	19	7	1386
spambase	57	2	2758
splice	60	3	1914
twonorm	20	2	4440
winequality-white	11	11	2938
balance	4	3	375
car	6	4	1036
flare	11	6	639
led7digit	7	10	300
pima	8	2	460
tic-tac-toe	9	2	574
wdbc	30	2	341
winequality-red	11	11	959
yeast	8	10	890

2.5.2.2 Results

Table 2.5 gives the comparisons between CL and MMHC with respect to classification performance. It can be seen that CL and MMHC show nearly the same classification performance with respect to ACC, and the number of winning datasets of CL and that of MMHC are similar. We conducted the *Student's t-test* with significance level $\alpha = 5\%$ and hypothesized-mean-difference set to 0 to determine if this difference is significant, and the result showed this difference was not statistically significant. As for CLL, the *Student's t-test* with significance level $\alpha = 5\%$ and hypothesized-mean-difference set to 0.02 showed that the classification performance of CL was significantly better than that of MMHC on average. However, the number of winning datasets of CL is close to that of MMHC.

Table 2.5: Classification results on two metrics

Metrics	NB	CL	MMHC
ACC	1	1.033(8)	1.034(5)
CLL	1	0.755(7)	0.772(9)

For ACC, larger ratios indicate better classification performance. For CLL, since the raw scores are non-positive, smaller ratios indicate better classification performance. In the parentheses we counted the number of winning datasets among the 28 datasets. i.e., on how many datasets CL (MMHC) produced better results than MMHC (CL), likewise, draws are not counted.

2.6 Discussion

At each curriculum stage, we learn a network over a subset of variables $\mathbf{X}_{(i)}$ conditioned on fixed values of the rest of the variables $\mathbf{X}'_{(i)}$. An obvious alternative is to learn a network over $\mathbf{X}_{(i)}$ while ignoring $\mathbf{X}'_{(i)}$. In the ideal case, the subnet learned by this approach would have exactly one type of discrepancy from the true Bayesian network: it would contain extra edges between variables in $\mathbf{X}_{(i)}$ that cannot be d-separated in the true Bayesian network without fixing certain variables in $\mathbf{X}'_{(i)}$. In this alternative approach, the scoring function of the subnet can be computed much faster than in our original algorithm. This is because we no longer have to partition the data based on the values of $\mathbf{X}'_{(i)}$ and hence only need to compute a single score over all the data samples. However, the theoretical guarantees given in Theorem 2.1 and 2.2

no longer hold with this alternative approach and counter-examples can be shown to exist. In addition, our experiments showed that this approach resulted in worse overall learning accuracy than our original algorithm.

2.7 Conclusion

In this chapter, we proposed a novel heuristic algorithm for Bayesian network structure learning. Our algorithm takes advantage of the idea of curriculum learning and learns the Bayesian network structure by stages. At each stage a subnet is learned over a selected subset of the random variables conditioned on fixed values of the rest of the variables. The selected subset grows with stages and eventually includes all the variables. We designed a new scoring function for curriculum learning that tailors the standard Bayesian scoring function to utilize all the training data and to alleviate overfitting. We constructed an incremental curriculum based on mutual information between variables. We also prove theoretically that our approach to learning Bayesian networks satisfies the desired property of curriculum learning that each intermediate target should be closer to the subsequent targets than any of its predecessors in the sequence based on both a structural and a distribution distance measures. The experimental results showed that not only did our algorithm outperform the state-of-the-art MMHC algorithm in recovering Bayesian network structures, but it also showed better classification performance when the learned Bayesian networks are used as classifiers.

CHAPTER 3. FINDING THE K-BEST EQUIVALENCE CLASSES FOR MODEL AVERAGING

In chapter 2, we discussed how we learn a good Bayesian network from the data and then use this single model for classification and inference. In some situations, learning a single optimal DAG is not sufficient - a single DAG is subject to noise and other idiosyncrasies in the data. As such, a data analyst would want to be aware of other likely DAGs. Hence, a number of algorithms have been proposed to enumerate the k-best DAGs from a complete dataset (Tian et al., 2010; Bartlett and Cussens, 2013).

There is a fundamental inefficiency in enumerating the k-best DAGs, namely that any given DAG may be Markov equivalent to many other DAGs, which are all equally expressive in terms of representing probability distributions. Thus, by enumerating DAGs, one may spend a significant amount of effort in enumerating redundant Bayesian networks.

In this chapter, we develop an algorithm called *kBestEC* to directly enumerate the k-best equivalence classes (ECs) of Bayesian networks. We show that our algorithm is significantly more efficient than the previous algorithm that directly finds the k-best DAGs (Tian et al., 2010). Moreover, we demonstrate our algorithm on the tasks of Bayesian model averaging (BMA) and causal structure discovery.

3.1 Preliminaries

In the problem of learning Bayesian networks from a data set D, we seek a DAG over the set of nodes indexed by $V = \{1, ..., n\}$ that best explains the data D, evaluated by some scoring function, e.g., $\ln P(G, D)$. In this work, we assume decomposable score such that

$$score(G : D) = \sum\nolimits_{v \in V} score_v(Pa_v^G : D), \tag{3.1}$$

where $score(G : D)$ will be written as $score(G)$ for short in the following discussion. Next we give a few of definitions and theorems that describe some additional semantics and properties of Bayesian networks.

Definition 3.1. *A v-structure in a DAG G is an ordered triple of nodes (u, v, w) such that G contains the directed edges $u \to v$ and $w \to v$ and u and w are not adjacent in G.*

Theorem 3.1. *(Verma and Pearl, 1990) Two DAGs G_1 and G_2 are equivalent if and only if they have the same skeleton and the same v-structures.*

Definition 3.2 (Score Equivalence)**.** *Let $score(G)$ be some scoring function that is decomposable. We say that it satisfies score equivalence if for any two equivalent DAGs G_1 and G_2 we have $score(G_1) = score(G_2)$ for any data set D.*

Score equivalence is the nature of several common scoring functions such as MDL, BDe and BIC. As a result, the set of equivalent DAGs are indistinguishable by these scoring functions. Thus, our goal is to find "a best", instead of "the best". However, finding a best DAG is NP-hard (Chickering, 1996). Recently, a family of DP algorithms have been developed to find a optimal DAG in time $O(n2^n)$ and space $O(2^n)$ (Singh and Moore, 2005; Silander and Myllymäki, 2006). The central idea exploits the fact that a DAG must have a sink s. Considering any $s \in V$, the best DAG over V with s as sink can be constructed by piecing together the best DAG $G^*_{V\setminus\{s\}}$ over $V \setminus \{s\}$ and the best parent set $Pa^*_s \subseteq V \setminus \{s\}$ assuming $G^*_{V\setminus\{s\}}$ is already known. Then we choose the best sink s that optimizes this construction. Applying this idea to all $W \subseteq V$ results in a DP algorithm that finds the best DAG for all 2^n possible W recursively. Figure **??** gives an example of the DP algorithm operating on a four-variable problem.

Later, Tian et al. (2010) generalized the algorithm (we will call it *kBestDAG* algorithm) to recursively find the k-best DAGs and proposed to make inference by averaging over these DAGs. Instead of considering a single best DAG, their algorithm maintains a list of k-best DAGs for each node set $W \subseteq V$. However, these k-best DAGs are redundant in the sense that they coalesce into only a fraction k of ECs and from one DAG we can efficiently infer other members in the same EC. Thus, it is desirable if we are able to directly find the k-best ECs. In next section, we present such an algorithm.

3.2 Finding the k-best Equivalence Classes of Bayesian Networks

3.2.1 Algorithm

The following definitions will be useful in the development of our algorithm.

Definition 3.3 (Score for sub-graph $G_W, W \subseteq V$)**.** *For any decomposable score, define* $score(G_W) = \sum_{v \in W} score_v(Pa_v^{G_W})$ *for any DAG* G_W *over any node set* $W \subseteq V$*, where* $Pa_v^{G_W}$ *is the parent set of* v *in* G_W*.*

Definition 3.4 (*Graph growth operator* $\oplus$)**.** *For any* G_W*,* $v \in V \setminus W$*,* $Pa_v \subseteq W$*, define* $G_{W \cup \{v\}} = G_W \oplus Pa_v$ *as an operation growing* G_W *to* $G_{W \cup \{v\}}$ *s.t.* $G_{W \cup \{v\}}$ *contains all edges in* G_W *and the directed edges from* Pa_v *to* v*.*

Lemma 3.1. *For any decomposable score function that satisfies score equivalence, we have* $score(G_W) = score(G'_W)$ *if* G_W *and* G'_W *are equivalent over node set* $W \subseteq V$*.*

The proof of Lemma 3.1 is given in the Appendix A.1. Lemma 3.1 says the score equivalence actually holds for DAGs over any subset $W \subseteq V$. This property allows us to recursively construct top equivalence classes over all $W \subseteq V$.

Now we outline the algorithm for finding the k-best equivalence classes given in Algorithm 3.1. It has three logical steps:

1. Compute the family scores $Score_v(Pa_v)$ for all $n2^{n-1}$ (v, Pa_v) pairs (lines 1–3);
2. Find the k-best parent sets in candidate set C for all $C \subseteq V \setminus \{v\}$ for all $v \in V$ (lines 4–6);
3. Recursively find the k-best equivalence classes over all node sets $W \subseteq V$ (in lexicographic order) (lines 7–13).

The first two steps follow naturally from those steps in (Silander and Myllymäki, 2006) and (Tian et al., 2010) and we will use their algorithms. Figure 3.1 gives an example of the algorithm operating on a four-variable problem $\{X_1, X_2, X_3, X_4\}$. Figure 3.1a shows how to recursively find the 2-best parent sets of the variable X_4 in all candidate sets.

Algorithm 3.1 Finding the k-best Equivalence Classes

1: **for** all $v \in V$ **do**
2: Compute $score_v(Pa_v)$ for all $Pa_v \subseteq V \setminus \{v\}$.
3: **end for**
4: **for** all $v \in V$ **do**
5: Find the k-best parent sets $\{bestPa_v(C,i), i = 1, ..., k\}$ in parent candidate set C for all $C \subseteq V \setminus \{v\}$ recursively.
6: **end for**
7: **for** all $W \subseteq V$ in lexicographic order **do**
8: A priority queue $bestDAGs(W)$ with size limit of k, initialized to $\emptyset$. The elements in $bestDAGs(W)$ is denoted by $G^i_W, i \in \{1, ..., k\}$.
9: **for** all $s \in W$ **do**
10: Find the k-best $G^1_{W,s}, ..., G^k_{W,s}$ with s as a sink from $\{G^i_{W\setminus\{s\}} \oplus bestPa_s(W \setminus \{s\}, j) : i = 1, ..., k, j = 1, ..., k\}$.
11: For all $i \in \{1, ..., k\}$, insert $G^i_{W,s}$ into queue $bestDAGs(W)$ if $score(G^i_{W,s}) > min\{score(G^i_W), i = 1, ..., k\}$ and $G^i_{W,s}$ is not equivalent to any DAG in $bestDAGs(W)$.
12: **end for**
13: **end for**
14: **return** $bestDAGs(V)$

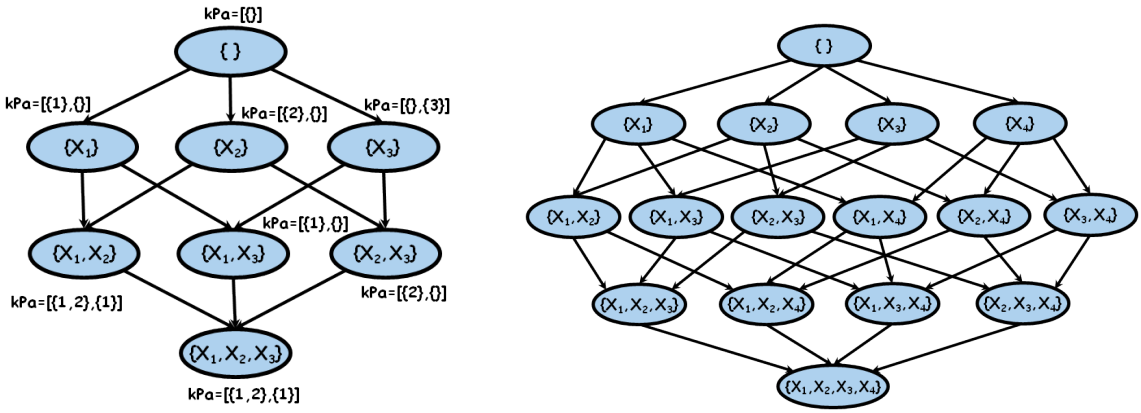
We will use the idea of DP to find the k-best equivalence classes recursively for all $W \subseteq V$, while the *kBestDAG* algorithm in (Tian et al., 2010) finds the k-best DAGs recursively. However working in the equivalence class space requires more careful treatment. It is not immediately clear that the idea of exploiting sink will work in the equivalence class space.

For a node set $W \subseteq V$, let EC^i_W, $i \in \{1, ..., k\}$ denote the top k equivalence classes over W. For each EC^i_W, we use a DAG over W, denoted as G^i_W, to represent the whole equivalence class.[1] For each $W \subseteq V$, we keep track of k DAGs, $G^1_W, ..., G^k_W$, each of them comes from one of the top k equivalence classes. Now assume we have identified such k-best ECs $G^1_{W\setminus\{s\}}, ..., G^k_{W\setminus\{s\}}$ for all $s \in W$. Finding the k-best ECs $G^1_W, ..., G^k_W$ for W takes two sub-steps:

3a. For each $s \in W$, identify the k-best ECs $G^1_{W,s}, ..., G^k_{W,s}$ over W with s as a sink (line 10 in Algorithm 3.1).

3b. Let $G^1_W, ..., G^k_W$ be the k-best nonequivalent DAGs among $\cup_{s\in W}\{k$-best ECs $G^1_{W,s}, ..., G^k_{W,s}$ over W with s as a sink$\}$ (line 11 in Algorithm 3.1).

[1] An alternative way to represent a EC is called completed partially DAG (CPDAG), consisting of a directed edge for every compelled edge and an undirected edge for every reversible edge in the EC (Chickering, 2002a). We choose DAG over CPDAG because: (1) encoding a DAG is space more efficient than encoding a CPDAG, which makes significant difference when we have to keep $k2^n$ networks in memory; (2) growing a DAG using $\oplus$ results in a valid DAG while growing CPDAG using $\oplus$ results in a PDAG which need be converted to a CPDAG with extra effort.



(a) Parent graph for X_4 (b) Order graph

Figure 3.1: An illustrative example of the DP algorithm operating on a four-variable problem ($\{X_1, X_2, X_3, X_4\}$). (a) An illustrative example for recursively finding the 2-best parent sets of variable X_4 in all possible candidate sets. (b)The order graph that illustrates the dependence structure and processing order of all subproblems in the DP algorithm.

In 3a, to find the k-best ECs $G^1_{W,s}, ..., G^k_{W,s}$, let $bestPa_s(C, j)$ denote the j-th best parent set for node s in the set of candidate parents C. Define function $value_{W,s}(i, j)$ by

$$value_{W,s}(i, j) = score(G^i_{W\setminus\{s\}}) + score_s(bestPa_s(W \setminus \{s\}, j)).$$

We can find the k-best scores among $\{value_{W,s}(i, j) : i, j \in \{1, ..., k\}\}$ by performing a best-first search with root node $(1, 1)$ and children of (i, j) being $(i + 1, j)$ and $(i, j + 1)$, as suggested by Tian et al. (2010). Let $G^1_{W,s}, ..., G^k_{W,s}$ denote the k DAGs $G^i_{W\setminus\{s\}} \oplus bestPa_s(W \setminus \{s\}, j)$ corresponding to the k-best scores. Now do they represent the k-best ECs? In other words, can some of these DAGs be equivalent to each other, or are there other DAGs having better scores than these DAGs? One concern is that in constructing $G^1_{W,s}, ..., G^k_{W,s}$ we only use one representative DAG $G^i_{W\setminus\{s\}}$ from its corresponding EC. Is it safe to ignore other DAGs equivalent to $G^i_{W\setminus\{s\}}$? The following theorem guarantees that $G^1_{W,s}, ..., G^k_{W,s}$ indeed represent the k-best ECs.

Theorem 3.2. *The k DAGs corresponding to the k-best scores output by the best-first search represent the k-best ECs over W with s as a sink.*

The proof of Theorem 3.2 is given in Appendix A.1.

After 3a, we have the k-best ECs over W with s as a sink for each $s \in W$.[2] In 3b, we identify $G^1_W, ..., G^k_W$ as the k-best DAGs from $\cup_{s \in W}\{k\text{-best } G^1_{W,s}, ..., G^k_{W,s}$ over W with s as a sink$\}$ that are mutually nonequivalent. For this purpose, we need explicitly check the equivalence of two DAGs if they are constructed from distinct sink s, s'. We first compare the scores. If the scores are not equal, two DAGs are nonequivalent. Otherwise, we need check whether they are equivalent. The detailed algorithm for checking the equivalence of two DAGs is in Appendix A.2.

Theorem 3.3. *The k DAGs $G^1_V, ..., G^k_V$ output by Algorithm 3.1 represent the k-best ECs over V.*

Proof. For each $W \subseteq V$, $\{G_W$ over $W\}= \cup_{s \in W}$ $\{G_W$ over W with s as a sink$\}$, therefore the k-best nonequivalent DAGs over W are the k-best among $\cup_{s \in W}\{k$-best ECs $G^1_{W,s}, ..., G^k_{W,s}$ over W with s as a sink$\}$. Thus, for each $W \subseteq V$, $G^1_W, ..., G^k_W$ obtained from Step 3b represent the k-best ECs over W. By induction, $G^1_V, ..., G^k_V$ output from Algorithm 3.1 represent the k-best ECs over V. □

An example of *kBestEC* to find the two best ECs over four variables $\{X_1, X_2, X_3, X_4\}$ is given in Figure 3.2. The DAGs inside each blue node are two DAGs representing the two best ECs over the corresponding subset W. Each blue node has an outgoing arc connected to a white node inside which are the two best DAGs over $W \cup \{s\}$ with s as sink. This corresponds to line 10 in Algorithm 3.1. Each blue node has several incoming arcs from corresponding white nodes. This process finds the two best DAGs over $W \cup \{s\}$ by merging the DAGs from previous step. This corresponds to line 11 in Algorithm 3.1. Note that when there are several equivalent DAGs, we keep one while discarding the others.

3.2.2 Characterization of Time and Space Complexity

Now we give a theoretical discussion on the run-time and space complexity of the algorithm. Step 1 takes $O(n2^{n-1})$ time and $O(2^{n-1})$ space. Step 2 takes $O(k \log k(n-1)2^{n-2})$ time and

[2] There may be less than k such DAGs for W when $|W|$ is small, but the number reaches k very rapidly as W grows.

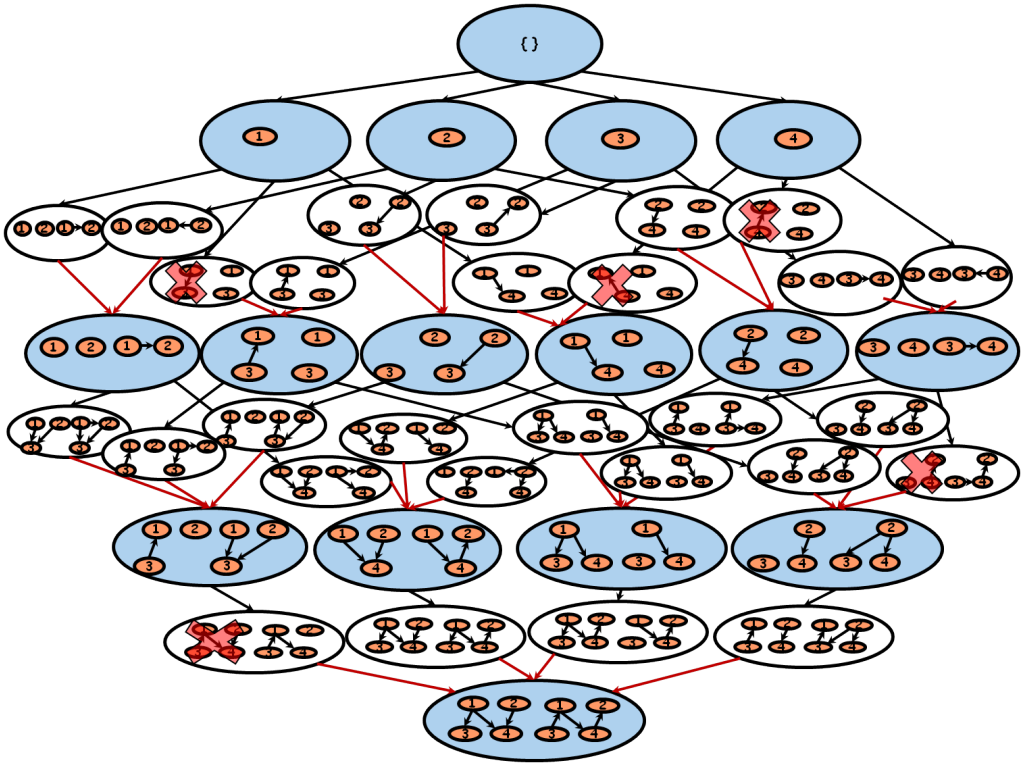


Figure 3.2: Finding the k-best ECs ($k = 2$) over $\{X_1, X_2, X_3, X_4\}$ by DP.

$O(k2^{n-1})$ space in the worst case. Doing a best-first search to find the k-best elements from space $\{(i, j) : i, j \in \{1, ..., k\}\}$ takes $O(k \log k)$ time. Checking the equivalence of two DAGs has a worst-case run-time of $O(|W|d_W^2)$, where d_W is the maximum size of parents in G_W and G'_W. Thus, the worst-case run-time for step 3 is $\sum_{|W|=1}^{n} \binom{n}{|W|} |W|(k \log k + k|W|d_W^2) = O(n2^{n-1}k(\log k + \frac{nd^2}{2}))$, where d is the maximum size of the parents computed in Step 2.[3] The worst space complexity is $O(k2^n)$ since we have to memorize no more than k DAGs for each $W \subseteq V$.[4]

For the same k, step 3 for finding the k-best ECs is $O(\frac{\log k + nd^2/2}{\log k + nd/2})$ times slower in the worst case than step 3 in *kBestDAG* for finding the k-best DAGs. Thus, *kBestEC* has slightly larger time complexity than *kBestDAG*. Both algorithms have the same space requirement.

[3] Checking whether two DAGs G_W and G'_W are the same has a run-time of $O(|W|d_W)$. Therefore the run-time for *kBestDAG* algorithm is $O(n2^{n-1}k(\log k + \frac{nd}{2}))$.

[4] We say worst space because for small W's, there may exist less than k equivalence classes.

3.3 Bayesian Model Averaging Using the k-best Equivalence Classes

We have presented an algorithm to obtain the k-best DAGs $G_V^1, ..., G_V^k$ representing the k-best equivalence classes $EC_V^1, ..., EC_V^k$. One application of our algorithm is to compute the posterior of hypothesis of interests with Bayesian model averaging (BMA). If the application is to evaluate class-invariant structural features such as Markov blanket or to predict new observations, the problem can generally be formulated as computing the posterior of the hypothesis h by

$$\hat{P}(h|D) = \frac{\sum_{i=1}^{k} w_i P(h|G_V^i, D) P(G_V^i, D)}{\sum_{i=1}^{k} w_i P(G_V^i, D)}, \tag{3.2}$$

where w_i is a weight we assign to each equivalence class EC_V^i. For example, if we want to treat each equivalence class as a single statistical model (Madigan et al., 1996; Castelo and Kocka, 2003), we simply set $w_i = 1$. If we'd like model averaging over original DAG space, we set $w_i = |EC_V^i|$, i.e, the number of DAGs in equivalence class EC_V^i.

If the application is to evaluate structural features such as an arrow $u \to v$ or a path $u \rightsquigarrow v$ that are not necessarily class-invariant, we have to enumerate the DAGs in each equivalence class in order to compute the posterior

$$\hat{P}(h|D) = \frac{\sum_{i=1}^{k} P(G_V^i, D) \sum_{G \in EC_V^i} P(h|G, D)}{\sum_{i=1}^{k} |EC_V^i| P(G_V^i, D)}. \tag{3.3}$$

Algorithm 3.2 sketches an algorithm to enumerate all DAGs in an equivalence class and to compute $|EC_V^i|$ in the mean time. Given a DAG G_V, we first determine the set of reversible edges, i.e., their directions vary among the equivalent DAGs (line 2). Chickering (1995) provided a $O(|E_{G_V}|)$ algorithm to find all compelled edges, i.e, their directions are invariant among the DAGs in an EC. We slightly modified this algorithm so that it outputs the set of reversible edges REV in G_V. All possible DAGs equivalent to G_V can be enumerated by reversing all possible edge combinations in REV. If the generated "DAG" passes the check of acyclicity and v-structures, it is a DAG equivalent to G_V. The overall algorithm takes $O((|V| + |E_{G_V}| + |E_{G_V}|^2)2^{|REV|})$ in the worst case. Note here we implemented a straightforward algorithm for enumerating all DAGs in an EC. Its run-time is negligible compared with the time for finding the k-best ECs due to the fact that the number of DAGs in an EC is pretty small.

Algorithm 3.2 *EnumEquivalentDAGs*(G_V)

```
1: list ← {G_V}
2: REV ← FindReversibleEdges(G_V)
3: for each subset CE ⊆ REV do
4:     Construct a new G'_V by reversing edges CE in G_V
5:     if CheckAcyclicity(G'_V)= true then
6:         flag ←true
7:         for each v participating in some edge of CE do
8:             if CheckVStruc(v, G_V, G'_V) = false then
9:                 flag ←false and break
10:            end if
11:        end for
12:        if flag = true then
13:            list.add(G'_V)
14:        end if
15:    end if
16: end for
17: return list
```

3.4 Experiments

We implemented Algorithm 3.1 in C++.[5]. To evaluate its performance, we consider the problem of computing the posteriors for all $n(n-1)$ possible directed edges using Equation 3.3 by enumerating all DAGs in each EC. We used BDe score (Heckerman and Chickering, 1995) for $score_i(Pa_i)$ with a uniform structure prior $P(G)$ and equivalent sample size 1. We compare the performances of our *kBestEC* algorithm with the *kBestDAG* algorithm, in terms of run-time, memory usage and quality of approximation. For approximation quality, we define cumulative posterior probability density of the set of DAGs in $\mathcal{G}$ used to perform model averaging by

$$\Delta = \sum_{G\in\mathcal{G}} P(G|D) = \frac{\sum_{G\in\mathcal{G}} P(G,D)}{P(D)}. \tag{3.4}$$

We used the algorithm from (Tian and He, 2009) to compute the exact $P(D)$ value. Note that $\Delta \leq 1$ and the larger of Δ, the closer of the estimation to the full Bayesian model averaging. In practice, it is often reasonable to make predictions using a collection of the best models discarding other models that predict the data far less well, even though the large amount of models with very small posteriors may contribute substantially to the sum such that Δ is much smaller than 1 (Madigan and Raftery, 1994). Therefore, we introduce another measure for the quality of estimation. We define the relative ratio of the posterior probability

[5] *kBestEC* is available at http://www.cs.iastate.edu/~jtian/Software/AAAI-14-yetian/KBestEC.htm

of the MAP structure G_{MAP} over the posterior of the worst structure G_{MIN} in the k-best ECs or DAGs by

$$\lambda = \frac{P(G_{MAP}|D)}{P(G_{MIN}|D)} = \frac{P(G_{MAP}, D)}{P(G_{MIN}, D)}. \tag{3.5}$$

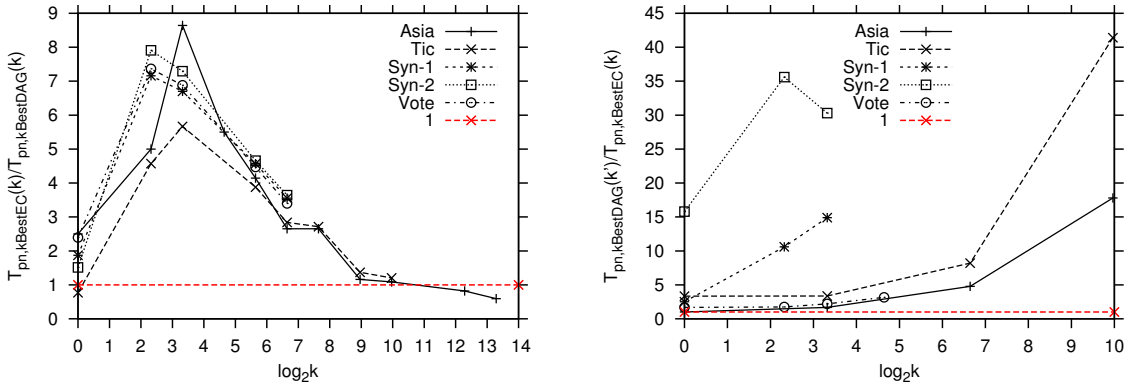
Note that both Δ and λ measures were used in (Tian et al., 2010).

3.4.1 kBestEC v.s. kBestDAG

We tested both algorithms on datasets from the UCI Machine Learning Repository as well as several synthetic datasets. All experiments were performed on a desktop with 2.4 GHz Intel Duo CPU and 4 GB of memory. The results are presented in Table 3.1. Besides k, Δ and λ, we list the number of variables n, sample size m, combined run-time T_{pn} for finding the k-best parent sets and finding the k-best ECs (or DAGs) (lines 4–14 in Algorithm 3.1), combined run-time T_e for enumerating DAGs (Algorithm 3.2) (0 for *kBestDAG* algorithm) and computing the posteriors, overall run-time T, total number of DAGs stored in memory $|\mathcal{G}_M|$, memory usage M (in MB), number of DAGs covered by the k-best ECs $|\mathcal{G}_k|$, and the average $\frac{|DAG|}{|EC|}$ ratio $\frac{|\mathcal{G}_k|}{k}$. All run-times are measured in seconds.

Our first observation is that, for all datasets, the running time T_e spent in enumerating all DAGs in k ECs is insignificant compared to the time for finding the k-best parent sets and ECs T_{pn} and the total time T. The total running time is dominated either by computing the local scores or by finding the k-best parent sets and the k-best ECs.

For the same k, BMA over k-best ECs has significantly better approximation quality than BMA over the k-best DAGs (see Δ values). This is straightforward since k ECs cover more than k DAGs and absorb more posterior probability density. $|\mathcal{G}_k|$ records the number of DAGs covered by the k-best ECs. Further, we see that *kBestEC* did spend more time for the same k as it requires extra overhead to respect equivalence. Both algorithms consume almost the same memory, which is consistent with the theory. An interesting observation is that *kBestEC* sometimes used slightly less memory than *kBestDAG* (see Asia $k = 1000$, $k = 1e4$, Tic $k = 1000$). This can be explained by comparing $|\mathcal{G}_M|$, the total number of DAGs stored in memory. *kBestEC* has smaller $|\mathcal{G}_M|$ than *kBestDAG*. This is because for small $W \subseteq V$, we usually have



(a) (b)

Figure 3.3: Comparison of *kbestEC* and*kbestDAG* on execution times. (a) Execution times of two algorithms to find k-best DAGs or ECs; (b) Execution times of two algorithms to achieve the same Δ values.

less than k distinct DAGs, and much less than k ECs to be stored. The effect is additive and in some cases causes big saving in both memory and time. For example, in case of Asia $k = 1e4$, $|\mathcal{G}_M|$ is significantly smaller for *kBestEC* than that for *kBestDAG*, such that *kBestEC* ($T = 539$ seconds) even ran faster than *kBestDAG* ($T = 896$ seconds).

A systematic comparison on T_{pn} of two algorithms for the same k is presented in Figure 3.3a. It plots the ratio $\frac{T_{pn,kBestEC}(k)}{T_{pn,kBestDAG}(k)}$ against $\log_2 k$ for all five data sets. A red dashed horizontal line is drawn for where the ratio is 1. All five curves peak between $k = 5$ and $k = 10$ and decrease rapidly as k becomes larger. This is because the ratio $\frac{T_{pn,kBestEC}(k)}{T_{pn,kBestDAG}(k)} = O(\frac{\log k + nd^2/2}{\log k + nd/2})$ decreases as k increases. Further, due to the same reason given above, as k increases, the number of the node sets W over which the number of distinct ECs is smaller than k increases. Because the run-time for computing each node set W is quadratic in the actual number of DAGs obtained for $W \setminus \{v\}$, *kBestEC* becomes more efficient.

Now we compare the two algorithms under the assumption that the same quality of approximation is achieved, i.e., they find the same number of DAGs, and therefore achieving the same Δ values. In order to achieve the same Δ as using k-best ECs, we have to run *kBestDAG* for a larger $k' = |\mathcal{G}_k|$ (the number of DAGs in the k-best ECs). With the same Δ, we observed that *kBestDAG* required significantly more time and memory. This is consistent with theoretical prediction of time ratio $O(\frac{k(\log k + nd^2/2)}{k'(\log k' + nd/2)})$ and space ratio $\frac{k}{k'}$. And for some Δ that *kBestEC*

Table 3.1: Performance comparison between *kBestEC* and *kBestDAG*

				kBestEC								***kBestDAG***						
Data	n	m	k	T_{pn}	T_e	T	$\lvert\mathcal{G}_M\rvert$	M	Δ	λ	$\lvert\mathcal{G}_k\rvert$	$\frac{\lvert\mathcal{G}_k\rvert}{k}$	T_{pn}	T_e	T	$\lvert\mathcal{G}_M\rvert$	M	Δ	λ
Asia	8	500	1	0.008	0.001	7.17	256	0.05	0.011	1	3	3	0.005	0	7.11	256	0.05	0.0036	1
			3										0.008	0	7.12	750	0.12	0.011	1
			10	0.06	0.01	7.20	2255	0.35	0.064	4.5	43	4.3	0.02	0.01	7.13	2283	0.36	0.022	2.3
			43										0.1	0.01	7.22	8502	1.31	0.064	4.5
			100	0.65	0.04	7.81	16981	2.61	0.225	17.2	467	4.67	0.27	0.02	7.4	17793	2.74	0.101	6.9
			467										3.1	0.07	10.3	77614	11.9	0.225	17.2
			1000	11.8	0.4	19.3	106631	16.5	0.525	129	4694	4.69	10.9	0.13	18.1	132503	20.4	0.316	28.2
			4694										209	0.65	217	476045	73.6	0.525	129
			1e+4	528	3.8	539	875329	135	0.805	1602	44864	4.49	887	1.28	896	969503	150	0.628	270
Tic	10	958	1	0.03	0.01	7.79	1024	0.16	0.059	1	7	7	0.04	0.01	7.81	1024	0.16	0.0084	1
			7										0.1	0.01	7.81	6922	1.06	0.059	1
			10	0.43	0.01	8.16	9777	1.50	0.563	1	67	6.7	0.06	0.01	7.9	9826	1.51	0.084	1
			67										1.44	0.01	9.17	59952	9.17	0.563	1
			100	5.18	0.07	13.1	86213	13.2	0.759	1005	673	6.73	2.4	0.02	10.3	87936	13.4	0.694	3.6
			673										42.6	0.07	51	545247	83.3	0.759	1005
			1000	102	0.73	111	677869	104	0.759	5.1e+7	7604	7.6	85.3	0.2	93.3	753873	115	0.759	2.2e+4
			7604										4226	0.8	4237	4967225	759	0.759	5.1e+7
Syn-1	15	100	1	1.2	0.02	18.2	32768	5.01	1.69e-5	1	4	4	0.8	0.01	18.2	32768	5.01	4.23e-6	1
			4										3.0	0.01	20	130919	20	1.69e-5	1
			10	26.2	0.06	43.2	326696	49.9	3.34e-4	1.9	114	11.4	10.4	0.01	27.5	326801	49.9	4.14e-5	1.1
			100	497	0.1	514	3224431	492	1.65e-3	4.4	1084	10.8	321	0.02	338	3230906	493	3.03e-4	1.9
			114										390	0.02	407	3681594	562	3.34e-4	1.9
Syn-2	15	200	1	0.96	0.04	24.7	32768	5.01	1.65e-3	1	13	13	0.77	0.01	24.5	32768	5.01	1.27e-4	1
			10	26.8	0.3	50.8	326696	49.9	0.0129	2.5	185	18.5	10.3	0.01	34.0	326801	49.9	1.27e-3	1
			13										15.2	0.01	39.0	424742	64.8	1.65e-3	1
			100	512	2.28	538	3224431	492	0.0483	10.3	1808	18.1	331	0.01	355	3230906	493	0.0081	1.9
			185										811	0.03	836	5967226	911	0.0129	2.5
Vote	17	435	1	6.21	0.09	172	131072	20.0	0.0125	1	3	3	3.8	0.01	172	131072	20.0	0.0042	1
			3										10.5	0.01	177	393180	60	0.0125	1
			10	122	1	289	1309470	200	0.0871	2.4	30	3	51	0.01	218	1309606	200	0.0376	1.3
			30										270	0.01	437	3924566	599	0.0871	2.4
			100	2684	5.34	2865	13031570	1988	0.302	10.8	318	3.18	1946	0.02	2150	13041226	1990	0.172	4.3

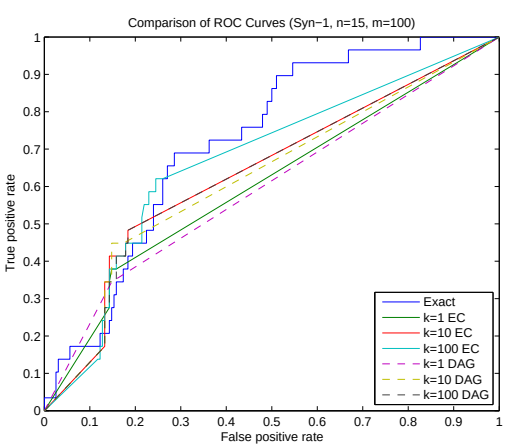
could easily achieve with the available resource, *kBestDAG* failed. In particular, for Syn-2 dataset, BMA over the top 100 ECs is equivalent to a BMA over the top 1808 DAGs. The former used only 492 MB memory, while the latter requires about 9 GB by estimation. Thus, *kBestEC* significantly outperformed *kBestDAG* in space and time usage to achieve the same quality of approximation.

A systematic comparison on T_{pn} of two algorithms when they find the same number of DAGs is presented in Figure 3.3b. It plots the ratio $\frac{T_{pn,kBestDAG}(k')}{T_{pn,kBestEC}(k)}$ for $k' = |\mathcal{G}_k|$, against $\log_2 k$ for all five data sets. A red dashed horizontal line is drawn for where the ratio is 1. The figure clearly shows that *kBestEC* is more efficient than *kBestDAG* in finding the same number of DAGs.

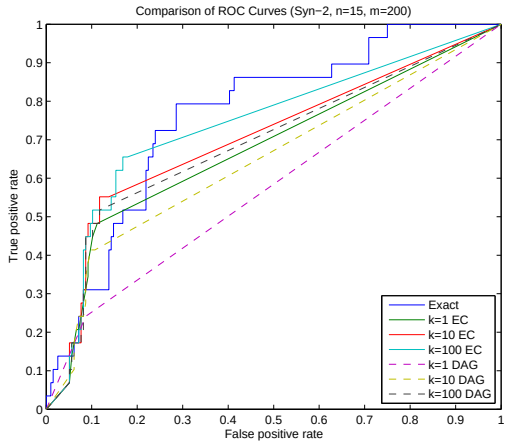
3.4.2 Structure Discovery

One important application of our algorithm is in (causal) structural discovery. We randomly generated several network structures of 15 variables and simulated datasets from them with sample size $m = 100, 200, 500$ respectively. We estimated the posteriors of all 210 possible edges by averaging over the corresponding k best ECs or DAGs ($k = 1, 10, 100$). For comparison, we also computed the edge posteriors using the exact method by Tian and He (2009). We then predicted the presence or absence of each edge based on its posterior (say, an edge $u \to v$ is present if $\hat{P}(u \to v|D) \geq 0.5$). The predictions were compared with ground truth and ROC curves were used to evaluate the predictive accuracy. Results are presented in Figure 3.4. It shows the accuracy for model averaging over the k-best ECs is significantly better than that over the k-best DAGs as expected.

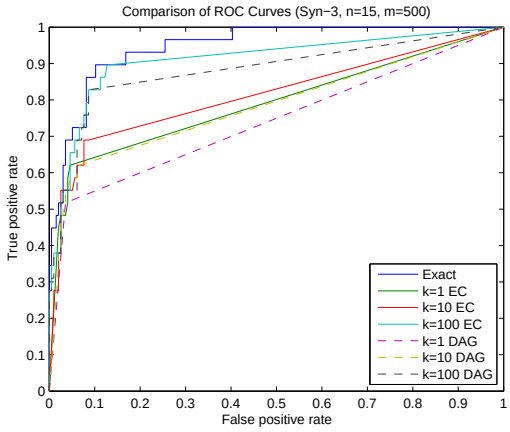
Another observation concerns about the reliability of using MAP model for structural inference. We first examine the λ value (Table 3.1). For Tic data set, the top 10 ECs are all equally probable. For data set Syn-1, the MAP equivalence class is only 1.9 times more probable than the 10-th best equivalence class, and only 4.4 times more probable than the 100-th best equivalence class. Similar results can be observed on Syn-2 and Vote data sets. This reflects that in many cases there are a significant number of distinct models explaining the data equally well and using MAP model for structure inference or causal reasoning is not reliable. Our algorithm



(a) n=15, m=100



(b) n=15, m=200



(c) n=15, m=500

Figure 3.4: Comparison of ROC curves for edge discovery.

will be a handy tool in understanding model structures in this kind of situation. A detailed comparison of the top 10 ECs for Tic data set is presented in Figure 3.5. It shows these 10 ECs agree only on one edge and disagree on other edges (even the skeleton). Further, most of the edges have probability below 0.5, indicating the high uncertainty on the network structure.

3.5 Discussion

Both *kBestDAG* and *kBestEC* are based on the DP algorithm. Recently, alternative approaches to finding an optimal Bayesian network have been proposed and shown being competitive or faster than the DP algorithm. These approaches include A* search (Yuan et al.,

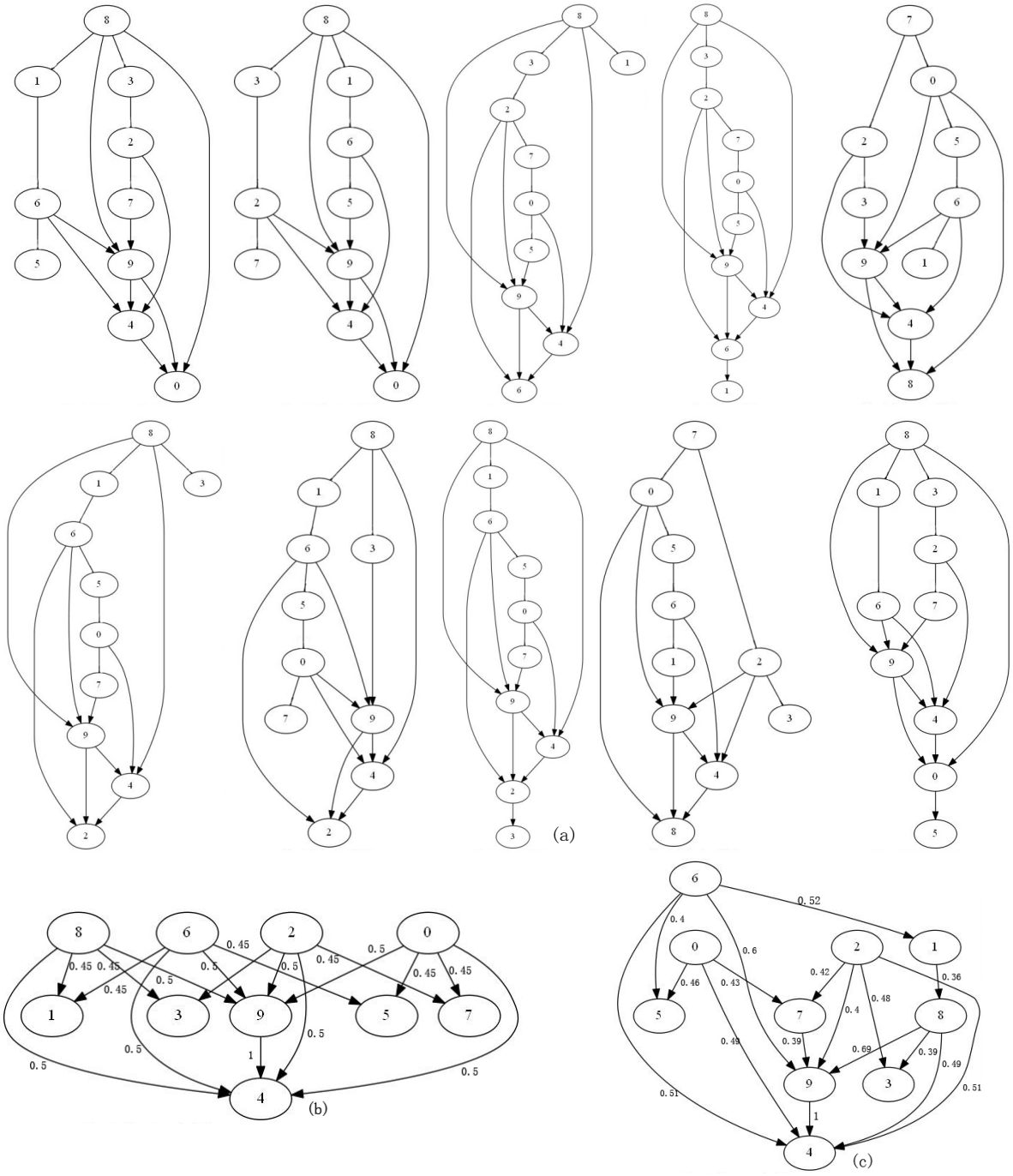


Figure 3.5: Structure discovery results on Tic data set. (a) The top 10 equally probable equivalence classes for Tic-Tac-Toe Data set. Each equivalence class is represented by a CPDAG where reversible edges are depicted as undirected edges, compelled edges are depicted as directed edges. (b) Network structure averaging over all DAGs. (c) Network structure averaging over top 10 ECs. The numbers besides the edges in (b) and (c) indicate the estimated posterior probabilities of edges. In these networks, we only include the most probable edges, i.e., edges whose posterior probabilities are greater than 0.4. We set this threshold such that the edges above this threshold do not form any directed cycles in the structure.

2011; Yuan and Malone, 2012; Malone and Yuan, 2012, 2013) and Integer Linear Programming (ILP) (Jaakkola et al., 2010; Cussens, 2011; Bartlett and Cussens, 2013). The A* search based algorithm URLearning formulates the learning problem as a shortest path finding problem and employs A* search algorithm to explore the search space. A potential future work is to explore the feasibility of generalizing the A* based algorithm to find the k-best DAGs or ECs. ILP based algorithm GOBNILP casts the structure learning problem as a linear program and solves it using the SCIP framework (Cussens, 2011). In such setting, it is possible to rule out specific DAGs with linear constraints. This allows GOBNILP to iteratively find the top k DAGs in deccreasing order of score (Bartlett and Cussens, 2013). Thus, another future work is to compare *kBestDAG*, *kBestEC* with GOBNILP in finding the k-best Bayesian networks.

3.6 Conclusion

In this chapter we developed an algorithm named *kBestEC* to find the k-best equivalence classes of Bayesian networks. It is the first approach to our knowledge for finding the k-best equivalence classes. We show that our algorithm is significantly more efficient than the previous *kBestDAG* algorithm that directly finds the k-best DAGs (Tian et al., 2010).

We tested *kBestEC* on the task of BMA to compute the posterior probabilities of edge features on several data sets from the UCI Machine Learning Repository as well as synthetic data sets. Our experiments showed that *kBestEC* significantly outperformed the *kBestDAG* algorithm in both time and space usages to achieve the same quality of approximation.

Our algorithm provides a useful tool for researchers interested in learning model structures or discovering causal structures. For example, biologists are interested in recovering gene regulation networks from data. Recovering the MAP network alone often does not give the full picture. There may exist a number of equally probable DAGs (or equivalence classes) with distinct structures when the amount of data is small relative to the size of the model. Our algorithm should be a very useful tool for understanding model structures in these situations by listing the most likely models and their relative likelihood.

CHAPTER 4. PARALLEL EXACT BAYESIAN EDGE LEARNING

In the Bayesian approach for structure discovery, one computes the posterior probability of a structural feature f by integrating over all possible DAGs G weighted by their corresponding posterior probabilities $P(G|D)$, i.e., $P(f|D) = \sum_G f(G)P(G|D)$, where f is an indicator function that $f(G) = 1$ if the feature is present in a DAG G and $f(G) = 0$ otherwise. Direct enumeration is infeasible in practice as the number of DAGs grows super-exponentially with the number of variables.

There are some cases where exact Bayesian learning is still tractable. Assuming an order-modular prior over DAGs and bounded in-degree, a dynamic programming (DP) algorithm proposed by Koivisto (2006a) can compute the posterior probabilities of any modular features, e.g., directed edges, in $O(n2^n)$ time and $O(2^n)$ space. Although this DP algorithm reduces the computation time from super-exponential to exponential, it is still insufficient because the largest problems it can solve on a typical desktop computer with a few GBs of memory do not exceed 25 variables. The memory usage is the bottleneck in practice.

In this chapter, we study how parallelism can be used to tackle the scalability problem of exact Bayesian structure discovery and present a parallel algorithm capable of computing the exact posterior probabilities of all possible edges with optimal parallel space efficiency and nearly optimal parallel time efficiency. We demonstrate the capability of our algorithm on datasets with up to 33 variables and its scalability on up to 2048 processors.

4.1 Introduction

Parallel computing aims to design systems and algorithms that use multiple processing elements simultaneously to solve a problem. It allows us to overcome the time and space

limitations by using supercomputers, which are usually equipped with thousands of processors and several terabytes of memory. If the computation steps in solving a problem are independent, the running time can be significantly reduced by parallelizing the execution of these independent steps on multiple processors. Certainly, this acceleration has theoretical upper bound. A widely used measure of the acceleration is *speedup*, defined as the ratio between the sequential running time (on one processor) and the parallel running time on p processors. Then in theory, $speedup \leq p$. That is, one can't achieve more than p times faster if p processors are used. The *speedup* will often be less than p as the parallel algorithm is bound to have some overhead in coordinating the actions of processors. Another measure of the performance of a parallel algorithm is *efficiency*, defined as the ratio between the sequential running time and the product of the number of processors used and the parallel running time. *Efficiency* measures how well the processors are utilized by the algorithm. Similarly, $efficiency \leq 1$. A parallel algorithm is said to be efficient if it involves the same order of work as performed by the best sequential algorithm. Most modern supercomputers implement a parallel model called the *distributed memory model*[1], where many processors are linked through high-speed connections and each processor has local memory directly attached to it. This type of supercomputers is scalable in terms of both the memory space and the number of processors. Thus, current research in parallel computing mainly use the *distributed memory model* for designing parallel algorithms.

As we have discussed in chapter 1, several parallel algorithms have already been developed for solving the *structure learning* problem, i.e., finding an optimal Bayesian network. In particular, Nikolova et al. (2009, 2013) described a parallel algorithm that can realize direct parallelization of the sequential DP algorithm in Ott et al. (2004) with optimal parallel efficiency. This algorithm is based on the observation that the subproblems constitute a lattice equivalent to an n-dimensional (n-D) hypercube, which has been proved to be a very powerful interconnection network topology used by most of modern parallel computer systems (Dally and Towles, 2004; Ananth et al., 2003; Loh et al., 2005). In the lattice formed by the DP subproblems, data exchange only happens between two adjacent nodes. In a hypercube inter-

[1]Another popular parallel model is the *shared memory model*, where a memory space is shared by all processors. This type of systems is typically very expensive and not scalable in terms of the memory size and the number of processors.

connection network, the neighbors communicate with each other much more efficiently than other pairs of nodes. By noting this, the parallel algorithm takes a direct mapping of the DP steps to the nodes of a hypercube, thus is communication-efficient. Further, this hypercube algorithm does not calculate redundant steps or scores. These two features render the implementation of the algorithm scalable on up to 2048 processors (Nikolova et al., 2013). Using 1024 processors with 512 MB memory per processor, they solved a problem with 30 variables in 1.5 hours.

In contrast, there is very limited work on scaling up *structure discovery*. To our knowledge, there is no parallel algorithm developed for computing the exact posterior probability of structural features. Although there are some similarities between the sequential DP algorithms for structure learning and structure discovery, they differ in some significant places. These differences prohibit the direct adaption of the existing parallel algorithms for *structure learning* to *structure discovery*. First, the DP algorithm for finding the optimal DAG involves only one DP procedure. All relevant scores for a subproblem are computed in one DP step, therefore can be computed on one processor. Thus, the mapping of subproblems to processors is very straightforward. However, the DP algorithm proposed by Koivisto (2006a) for computing the posterior probability of structural features involves several separate DP procedures, responsible for computing different scores. These DP procedures, though can be performed separately, rely on the completion of one another. Thus, it is a challenge to effectively coordinate the computations of these DP procedures in a parallel setup. Failure to do this may greatly harm the parallel efficiency. Second, the DP algorithm for computing the posterior probability of structural features involves two critical subtasks, each of which calls for a fast computation of a zeta transform variant. These two zeta transform variants require efficient parallel processing.

To fill up the gap, we develop a parallel algorithm to compute the exact posterior probability of substructures (e.g., edges) in Bayesian networks. Our algorithm realizes direct parallelization of the DP algorithm in (Koivisto, 2006a) with nearly perfect load-balancing and optimal parallel time and space efficiency, i.e., the time and space complexity per processor are $O(n2^{n-k})$ respectively, for p number of processors, where $k = log(p)$. Our parallel algorithm is an extension of Nikolova et al. (2009)'s hypercube algorithm to the *structure discovery* problem.

However, because of the difficulties discussed previously, our work goes beyond that by a significant margin. First, we adopt a delicate way to map the calculation of various scores to the processors such that large amount of data exchange between non-neighboring processors is avoided during the transition among the separate DP procedures. This manipulation significantly reduces the time spent in communication. Second, we develop novel parallel algorithms for two fast zeta transform variants. As zeta transforms are fundamental objects in several several combinatorial problems such as graph coloring (Koivisto, 2006b) and Steiner tree (Nederlof, 2009) and combinatorial tools like the fast subset convolution (Björklund et al., 2007), the parallel algorithms developed here would also benefit the researches outside the context of Bayesian networks.

The rest of this chapter is organized as follows. In section 4.2, we present some preliminaries of exact structure discovery in Bayesian networks and briefly review Koivisto (2006a)'s DP algorithm, upon which our parallel algorithm is based. In section 4.3, we present our parallel algorithm for computing the posterior probability of structural features and conduct a theoretical analysis on its run-time and space complexity. In section 4.4 , we empirically demonstrate the capability of our algorithm on a Dell PowerEdge C8220 supercomputer. Discussions and conclusions are presented in section 4.5.

4.2 Exact Bayesian Structure Discovery in Bayesian Networks

We first review the DP algorithm in (Koivisto and Sood, 2004) for computing the posteriors of modular structural features.

4.2.1 Computing Posteriors of Structural Features

A structural feature, e.g., an edge, is conveniently represented by an indicator function f such that $f(G)$ is 1 if the feature is present in G and 0 otherwise. In Bayesian approach, we are interested in computing the posterior probability $P(f|D)$ of the feature, which can be obtained by computing the joint probability $P(f, D)$ by

$$P(f, D) = \sum_{G} f(G)P(G, D). \tag{4.1}$$

Instead of directly summing over the super-exponential DAG space, Friedman and Koller (2003) proposed to work on the order space, which was demonstrated more efficient and convenient. Formally, an order $\prec$ is a linear order $(L_1, ..., L_n)$ on the index set V, where L_i specifies the predecessors of i in the order, i.e., $L_i = \{j : j \prec i\}$. We say that a DAG $G = (G_1, ..., G_n)$ is consistent with an order $\prec$ if $G_i \subseteq L_i$ for all i. By introducing the random variable $\prec$, $P(f, D)$ can be computed alternatively by

$$P(f, D) = \sum_{\prec} P(\prec) P(f, D|\prec). \tag{4.2}$$

Assume an *order modular prior* defined as follows: if G is consistent with $\prec$, then

$$P(\prec, G) = \prod_{i=1}^{n} q_i(L_i)\rho_i(G_i), \tag{4.3}$$

where each q_i and ρ_i is some function from the subsets of $V - \{i\}$ to the nonnegative reals. We will also make the standard assumptions on global and local parameter independence, and parameter modularity (Cooper and Herskovits, 1992). Further, in this paper, we consider only modular features, i.e, $f(G) = \prod_{i=1}^{n} f_i(G_i)$, where each $f_i(G_i)$ is an indicator function with values either 0 or 1. For example, an edge $u \to v$ can be represented by setting $f_v(G_v) = 1$ if and only if $u \in G_v$, and setting $f_i(G_i) = 1$ for all $i \neq v$. In addition, we assume the number of parents of each node is bounded by a constant d. With these assumptions, Koivisto and Sood (2004) showed that Equation 4.2 can be factorized as

$$P(f, D) = \sum_{\prec} \prod_{i=1}^{n} q_i(L_i) \sum_{G_i:\ G_i \subseteq L_i \text{ and } |G_i| \leq d} \rho_i(G_i) p(x_i|x_{G_i}, G_i) f_i(G_i), \tag{4.4}$$

where $p(x_i|x_{G_i}, G_i)$ is the local marginal likelihood for variable i, measuring the local goodness of G_i as the parents of i. For convenience, for each family (i, G_i), $i \in V$, $G_i \subseteq V - \{i\}$, we define

$$B_i(G_i) \equiv \rho_i(G_i) p(x_i|x_{G_i}, G_i) f_i(G_i). \tag{4.5}$$

Note that if we assume the bounded in-degree d, we only need to compute $B_i(G_i)$ for $G_i \subseteq V - \{i\}$ with $|G_i| \leq d$. Further, for all $i \in V$, $S \subseteq V - \{i\}$, define

$$A_i(S) \equiv q_i(S) \sum_{G_i:\ G_i \subseteq S \text{ and } |G_i| \leq d} B_i(G_i). \tag{4.6}$$

The sum on the right-hand side of Equation 4.6 is known as a variant of the *zeta transform* of B_i, evaluated at S. Now Equation 4.4 can be neatly written as

$$P(f, D) = \sum_{\prec} \prod_{i=1}^{n} A_i(L_i). \tag{4.7}$$

Koivisto and Sood (2004) showed that $P(f, D)$ can be computed by defining a recursive function F on all $S \subseteq V$,

$$F(S) \equiv \sum_{i \in S} A_i(S - \{i\}) F(S - \{i\}), \tag{4.8}$$

with the base case $F(\emptyset) \equiv 1$. Then $P(f, D) = F(V)$.

With this definition, $P(f, D) = F(V)$ can be computed efficiently with dynamic programming. Then the posterior probability of the feature f is obtained by $P(f|D) = P(f, D)/P(D)$, where $P(D)$ can be computed like $P(f, D)$ by simply setting all features $f_i(G_i) = 1$, i.e. $P(f = 1, D) = P(D)$.

Computing $B_i(G_i)$ scores for all $i \in V$, $|G_i| \leq d$ takes $O(n^{d+1})$ time.[2] For any $i \in V$, A_i scores can be computed in $O(d2^n)$ time with a technique called the *fast truncated upward zeta transform*[3] (Koivisto and Sood, 2004). The recursive computation of $F(V)$ takes $O(\sum_{i=0}^{n} i \cdot \binom{n}{i}) = O(n2^n)$ time. The total time for computing one feature (i.e., an edge) is therefore $O(n^{d+1} + nd2^n + n2^n) = O((d + 1)n2^n)$.

4.2.2 Computing Posterior Probabilities for All Edges

If the application is to compute the posteriors for all $n(n - 1)$ directed edges, we can run above algorithm separately for each edge. Then the time for computing all $n(n - 1)$ directed edges is $O((d + 1)n^3 2^n)$. Since the computations for different edges involve a large proportion of overlapping elements, a forward-backward algorithm was provided in (Koivisto, 2006a) to reduce the time to $O(2(d+1)n2^n)$. For all $S \subseteq V$, define a "backward function" recursively as

$$R(S) = \sum_{i \in S} A_i(V - S) R(S - \{i\}), \tag{4.9}$$

[2] We assume the computation of $p(x_i|x_{G_i}, G_i)$ takes $O(1)$ time here. However, it is usually proportional to the sample size m.

[3] It is called Möbius transform in (Koivisto and Sood, 2004; Koivisto, 2006a), but *zeta transform* is actually the correct term.

with the base case $R(\emptyset) = 1$. Then for any fixed node $v \in V$ (the endpoint of an edge) and $u \in V - \{v\}$, the joint distribution $P(u \to v, D)$ can be computed by

$$P(u \to v, D) = \sum_{G_v:\ u \in G_v \subseteq V - \{v\} \text{ and } |G_v| \leq d} B_v(G_v)\Gamma_v(G_v), \tag{4.10}$$

where for all $v \in V$, $G_v \subseteq V - \{v\}$

$$\Gamma_v(G_v) \equiv \sum_{S: G_v \subseteq S \subseteq V - \{v\}} q_v(S)F(S)R(V - \{v\} - S). \tag{4.11}$$

The sum on the right-hand side of Equation 4.11 is another variant of the *zeta transform*. Provided that B_i, A_i, F, R are precomputed with respect to $f \equiv 1$, for any endpoint node v, $\Gamma_v(G_v)$ can be computed in $O(d2^n)$ time for all $G_v \subseteq V - \{v\}$, $|G_v| \leq d$ with a technique called *fast downward zeta transform* (Koivisto, 2006a). To evaluate Equation 4.10 for a different $u \in V - \{v\}$, we only need to recompute the function B_v by changing only the function f_v. Thus, evaluating Equation 4.10 takes $O(n^d)$ time.

We then arrived at the following algorithm for computing the posteriors for all $n(n-1)$ edges. Let the functions B_i, A_i, Γ_i, F and R be defined with respect to the trivial feature $f \equiv 1$ ($f_i(G_i) = 1$ for all $i \in \{1, ..., n\}$ and $G_i \subseteq V - \{i\}$).

Algorithm 4.1 Compute posterior probabilities for all $n(n-1)$ edges by DP (Koivisto, 2006)

for all $i \in V$ and $S \subseteq V - \{i\}$ with $|S| \leq d$: compute $B_i(S)$.
for all $i \in V$ and $S \subseteq V - \{i\}$: compute $A_i(S)$.
for all $S \subseteq V$: compute $F(S)$ recursively.
for all $S \subseteq V$: compute $R(S)$ recursively.
for all $v \in V$ **do**
    for all $G_v \subseteq V - \{v\}$ with $|G_v| \leq d$: compute $\Gamma_v(G_v)$.
    for all $u \in V - \{v\}$ **do**
        Compute $P(u \to v, D) = \sum_{G_v:\ u \in G_v \subseteq V - \{v\} \text{ and } |G_v| \leq d} B_v(G_v)\Gamma_v(G_v)$
        Evaluate $P(u \to v|D) = P(u \to v, D)/F(V)$.
    end for
end for

Adding up the time for all steps, the total computation time for evaluating all $n(n-1)$ edges is $O(n^{d+1} + dn2^n + n2^n + n2^n + n^{d+2} + dn2^n) = O(2(d+1)n2^n)$.

4.3 Parallel Algorithm

We use the Algorithm 4.1 presented in section 4.2 as a base for parallelization. The computation of A functions corresponds to a variant of *zeta transform* which are computationally intensive. There is no known parallel algorithm for *zeta transform* (Rota, 1964). Here we design a novel parallel algorithm for it.[4]

The recursive computations of functions F and R over node sets $S \subseteq V$ are analogous to DP techniques in the algorithm for finding optimal BN in (Ott et al., 2004). Thus, it is possible to use the parallel techniques developed for the latter problem. In our algorithm, we adapt Nikolova et al. (2009)'s hypercube algorithm and use it as sub-routines to compute functions F and R. However, some difficulties prohibit the direct adaption. The computation in Algorithm 4.1 consists of several consecutive procedures, each of which is responsible for computing a particular function. The computations of these functions depend on one another. For example, computing F and R need A's being available. Note that all functions are evaluated over 2^n of subsets $S \subseteq V$. In the hypercube algorithm, these subsets are computed in different processors, thus stored locally. Generally, processors need exchange their scores in order to compute a new function. Thus, it is challenging to coordinate the computations of these functions to reduce the number of messages sent between the processors, particularly between those non-neighboring processors. In our algorithm, we adopt a delicate way to achieve this.

The computation of Γ functions corresponds to another variant of *zeta transform* for which we again design a novel parallel algorithm.

Finally, we integrate these techniques into a parallel algorithm capable of computing the posteriors $P(u \to v|D)$ for all $n(n-1)$ edges with nearly perfect load-balancing and optimal parallel efficiency.

To facilitate presentation, in section 4.3.1, we first describe an ideal case, where 2^n processors are available. In this case, we can directly map the 2^n of subsets $S \subseteq V$ to an n-dimensional hypercube computing cluster. In section 4.3.2, we then generalize the mapping to a k-dimensional hypercube with $k < n$.

[4]The B functions required for computing A functions are computed inside the algorithm for computing A functions.

4.3.1 n-D Hypercube Algorithm

In section 4.3.1.1, we first describe the parallel algorithms for computing functions F and R as they explain why we base our parallel algorithm on the hypercube model. We postpone the discussion of computing B, A scores to section 4.3.1.2.

4.3.1.1 Computing $F(S)$ and $R(S)$

The DP algorithm for computing functions F can be visualized as operating on the lattice $\mathcal{L}$ formed by the partial order "set inclusion" on the power set of V (see Figure 4.1). The lattice $\mathcal{L}$ is a directed graph (V', E'), where $V' = 2^V$ and $(T, S) \in E'$ if $T \subset S$ and $|S| = |T| + 1$. The lattice is naturally partitioned into levels, where level l $(0 \leq l \leq n)$ contains all subsets of size l. A node S at level l has l incoming edges from nodes $S - \{i\}$ for each $i \in S$, and $n - l$ outgoing edges to nodes $S \cup \{j\}$ for each $j \notin S$. By Equation 4.8, node S receives $A_i(S - \{i\}) \cdot F(S - \{i\})$ from each of its incoming edges and computes $F(S)$ by summing over l such scores. Assuming $A_j(S)$ for all $j \notin S$ are precomputed and available at node S, node S will compute $A_j(S) \cdot F(S)$ for all $j \notin S$ then send the scores to corresponding nodes. For example, $A_j(S) \cdot F(S)$ is sent to node $S \cup \{j\}$ so that it can be used for computing $F(S \cup \{j\})$. Each level in the lattice can be computed concurrently, with data flowing from one level to the next.

If each node in $\mathcal{L}$ is mapped to a processor in a computer cluster, the undirected version of $\mathcal{L}$ is equivalent to an n-dimensional (n-D) hypercube, a network topology used by most of modern parallel computer systems (Dally and Towles, 2004; Ananth et al., 2003; Loh et al., 2005). We encode a subset S by an n-bit string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Accordingly, we can use ω to denote the id of the processor that the subset S is mapped to. As lattice edges connect pairs of nodes whose n-bit string differ by one element, they naturally correspond to hypercube edges (Figure 4.1). This suggests an obvious parallelization on an n-D hypercube.

The n-D hypercube algorithm runs in $n + 1$ steps. Let $\mu(\omega)$ denote the number of 1's in ω. Each processor is active in only one time step – processor ω is active in time step $\mu(\omega)$. It receives one $A_i(S - \{i\}) \cdot F(S - \{i\})$ value from each of $\mu(\omega)$ neighbors obtained by inverting

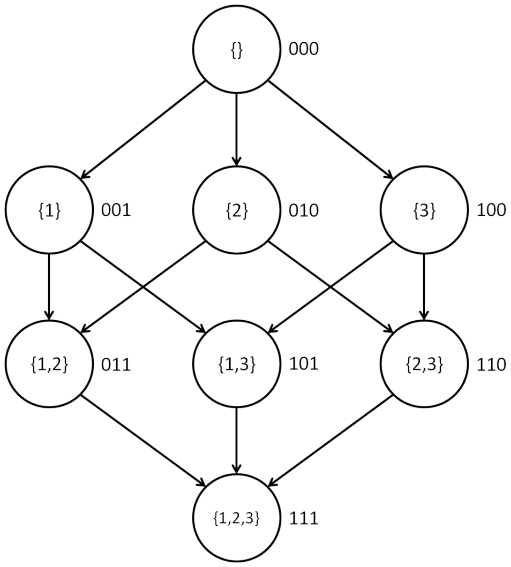


Figure 4.1: A lattice for a domain of size 3. The binary string labels on the right-hand side of each node show the correspondence with a 3-dimensional hypercube. B_i, A_i and function F for each subset S are computed at corresponding processor. The arrows show how the data flow between subproblems.

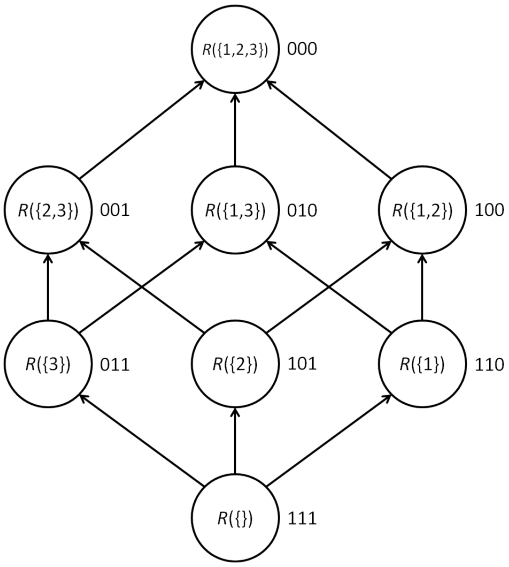


Figure 4.2: Map the computation of function $R(S)$ to the n-D hypercube. The binary string label on the right-hand side of each node denote the *id* of the processor. The arrows show how the data flow between subproblems.

one of its 1 bits to 0. It then computes its $F(S)$ function, computes $A_j(S) \cdot F(S)$ for all $j \notin S$ and sends them to its $n - \mu(\omega)$ neighbors obtained by inverting one of its 0 bits to 1. The run-time of step l is $O(l + n - l) = O(n)$. The parallel run-time for computing all F scores is $O(n^2)$ in total.

We can parallelize the computation of function R in the same manner. However, we have assumed $A_j(S)$ for all $j \notin S$ are available only at node S. To compute $R(S)$, node S need receive $A_i(V - S) \cdot R(S - \{i\})$ from its neighbors. However, $A_i(V - S)$ are available at neither node $S - \{i\}$ nor node S, but node $V - S$. Further, it is not a good idea either to have $F(S)$ and $R(S)$ at the same processor as each term of the summation in the computation of Γ scores requires different F and R (see Equation 4.11). To reduce message passing, we take a completely different mapping for computing R. The new mapping is illustrated in Figure 4.2. Note that $R(S)$ is computed at the processor where $F(V - S)$ is computed and all $A_i(V - S)$ are available. Processor ω receives one $R(S - \{i\})$ from each of its $n - \mu(\omega)$ neighbors obtained by inverting one of its 0 bits to 1. It then computes $R(S)$ by Equation 4.9 and sends it to all

its $\mu(\omega)$ neighbors obtained by inverting one of its 1 bits to 0. The processors in the hypercube operate in a bottom-up manner, e.g., starting from processor 111 and ending at processor 000. Similarly, the parallel run-time is $O(n^2)$.

4.3.1.2 Parallel Fast Zeta Transforms

In section 4.3.1.1, we have assumed $A_i(S)$ for all $i \notin S$ are precomputed at node S. For any $i \in V$, computing $A_i(S)$ for any subset $S \subseteq V - \{i\}$ requires the summation over all subsets of S with size no more than d (see Equation 4.6). If processors in the hypercube compute their A_i independently, the processor responsible for computing $A_i(V-\{i\})$ for all $i \in V$ takes $O(d2^{n-1})$ time. This certainly nullifies our effort of improving time complexity by parallel algorithm. In this section, we describe parallel algorithms with which all A_i (and Γ_v) scores can be computed on the n-D hypercube cluster in $O(n^2)$ time.

First, we give definitions for two variants of the well-known *zeta transform* (Kennes, 1992). Let $V = \{1, ..., n\}$. Let $s : 2^V \to \mathbb{R}$ be a mapping from the subsets of V onto the real numbers. Let $d \leq |V|$ be a positive integer.

Definition 4.1 (Truncated Upward Zeta Transform)**.** *(Koivisto and Sood, 2004) A function $t : 2^V \to \mathbb{R}$ is the truncated upward zeta transform of s if*

$$t(T) = \sum_{S \subseteq T : |S| \leq d} s(S), \text{ for all } T \subseteq V.$$

Definition 4.2 (Truncated Downward Zeta Transform)**.** *(Koivisto, 2006a) A function $t : 2^V \to \mathbb{R}$ is the truncated downward zeta transform of s if*

$$t(T) = \sum_{S : T \subseteq S \subseteq V} s(S), \text{ for all } T \subseteq V \text{ with } |T| \leq d.$$

It is easy to see that the function A_i for all $i \in V$ can be viewed as a case of the truncated *upward* zeta transform. Similarly, the function $\Gamma_v(G_v)$ can be viewed as a case of the truncated *downward* zeta transform.

Two techniques introduced in (Koivisto and Sood, 2004) and (Koivisto, 2006a) are able to realize both transforms in $O(d2^n)$ time, respectively. Here, we present the parallel versions of the two algorithms (see Algorithm 4.2 and Algorithm 4.3) that run on an n-D hypercube

computer cluster. The serial versions of the algorithms are given in (Koivisto and Sood, 2004) and (Koivisto, 2006a), respectively.

Algorithm 4.2 Parallel Truncated Upward Zeta Transform on n-D hypercube

Assumption: each subset $S \subseteq V$ is encoded by an n-bit string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Subset S is computed on processor with *id* ω.

1: On each , $t_0(S) \leftarrow s(S)$ for $|S| \leq d$ and $t_0(S) \leftarrow 0$ otherwise.
2: **for** $j \leftarrow 1$ to n **do**
3: **for** each processor ω s.t. $|S \cap \{j+1, ..., n\}| \leq d$ **do**
4: $t_j(S) \leftarrow 0$
5: **if** $|S \cap \{j, ..., n\}| \leq d$ **then**
6: $t_j(S) \leftarrow t_{j-1}(S)$
7: **end if**
8: **if** $j \in S$ **then**
9: Retrieve $t_{j-1}(S - \{j\})$ from processor $\omega' = \omega \oplus 2^{j-1}$.
10: $t_j(S) \leftarrow t_j(S) + t_{j-1}(S - \{j\})$
11: **end if**
12: **end for**
13: **end for**
14: **return** $t_n(S)$ on processor ω

Algorithm 4.3 Parallel Truncated Downward Zeta Transform on n-D hypercube

Assumption: each subset $S \subseteq V$ is encoded by an n-bit string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Subset S is computed on processor with *id* ω.

1: On each processor, $t_0(S) \leftarrow s(S)$.
2: **for** $j \leftarrow 1$ to n **do**
3: **for** each processor ω s.t. $|S \cap \{1, ..., j\}| \leq d$ **do**
4: $t_j(S) \leftarrow t_{j-1}(S)$
5: **if** $j \notin S$ **then**
6: Retrieve $t_{j-1}(S \cup \{j\})$ from processor $\omega' = \omega \oplus 2^{j-1}$.
7: $t_j(S) \leftarrow t_j(S) + t_{j-1}(S \cup \{j\})$
8: **end if**
9: **end for**
10: **end for**
11: **return** $t_n(S)$ on processor ω

By our definition in section 4.3.1.1, a subset S is encoded by an n-bit string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. In an n-D hypercube, ω is also used to denote the *id* of a processor. We can take this natural mapping so that each processor ω is responsible for the corresponding subset S. This forms the basic idea of the two parallel algorithms.

Algorithm 4.2 runs for $n + 1$ iterations. In each iteration, all 2^n processors operate on their $S \subseteq V$ concurrently (lines 3 to 12). In iteration j, before the computation starts, each processor ω with $\omega[j] = 0$ sends its $t_{j-1}(S)$ to its neighbor ω' obtained by inverting its $\omega[j]$ to

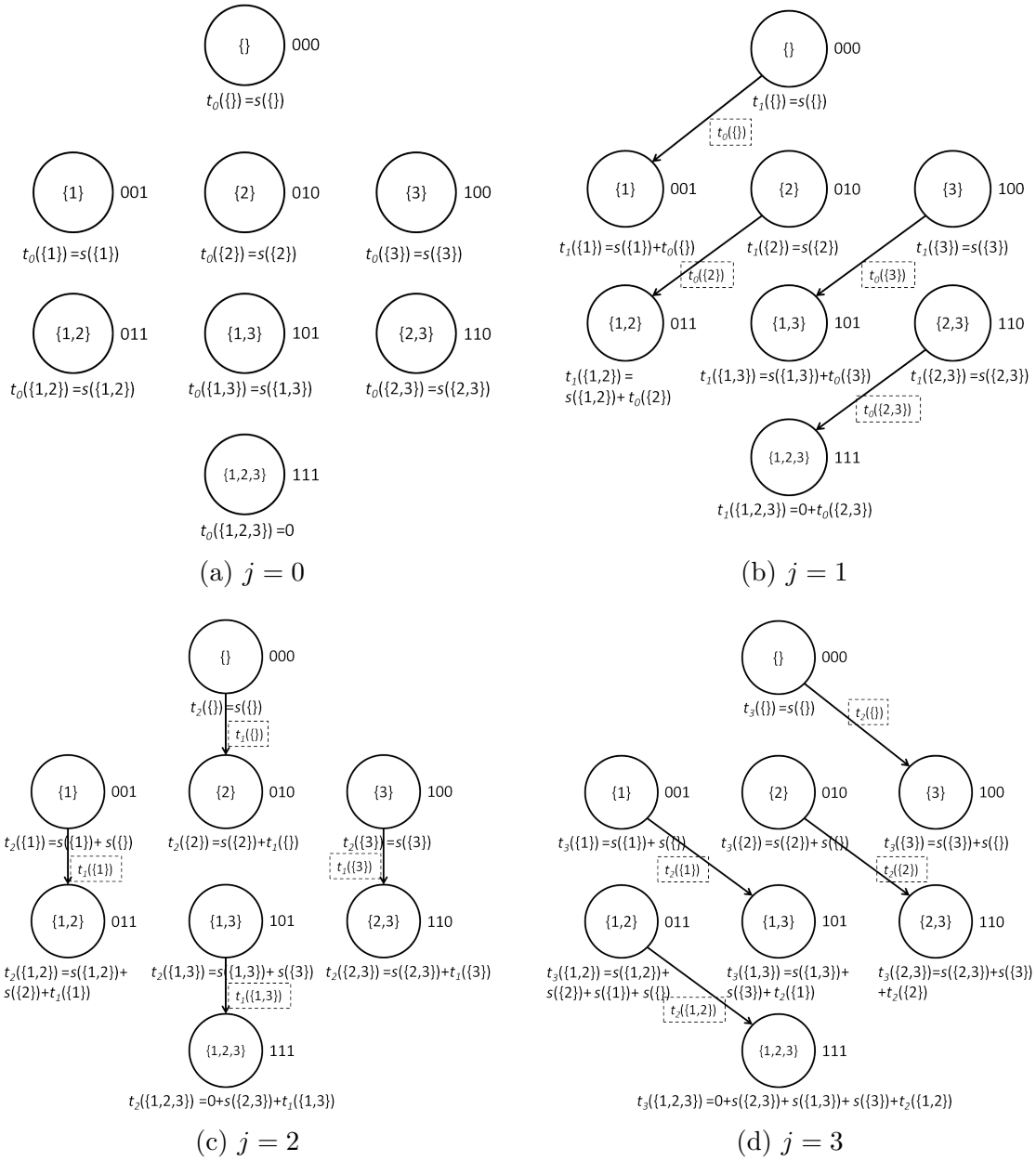


Figure 4.3: An illustrative example of parallel truncated upward zeta transform on n-D hypercube. In this case, $n = 3$, $d = 2$. The algorithm takes four iterations. Iterations 0, 1, 2 and 3 are show in (a), (b), (c) and (d), respectively. The functions in dashed boxes are the messages sending between the processors. In each iteration, the computed $t_j(S)$ is shown underneath each processor.

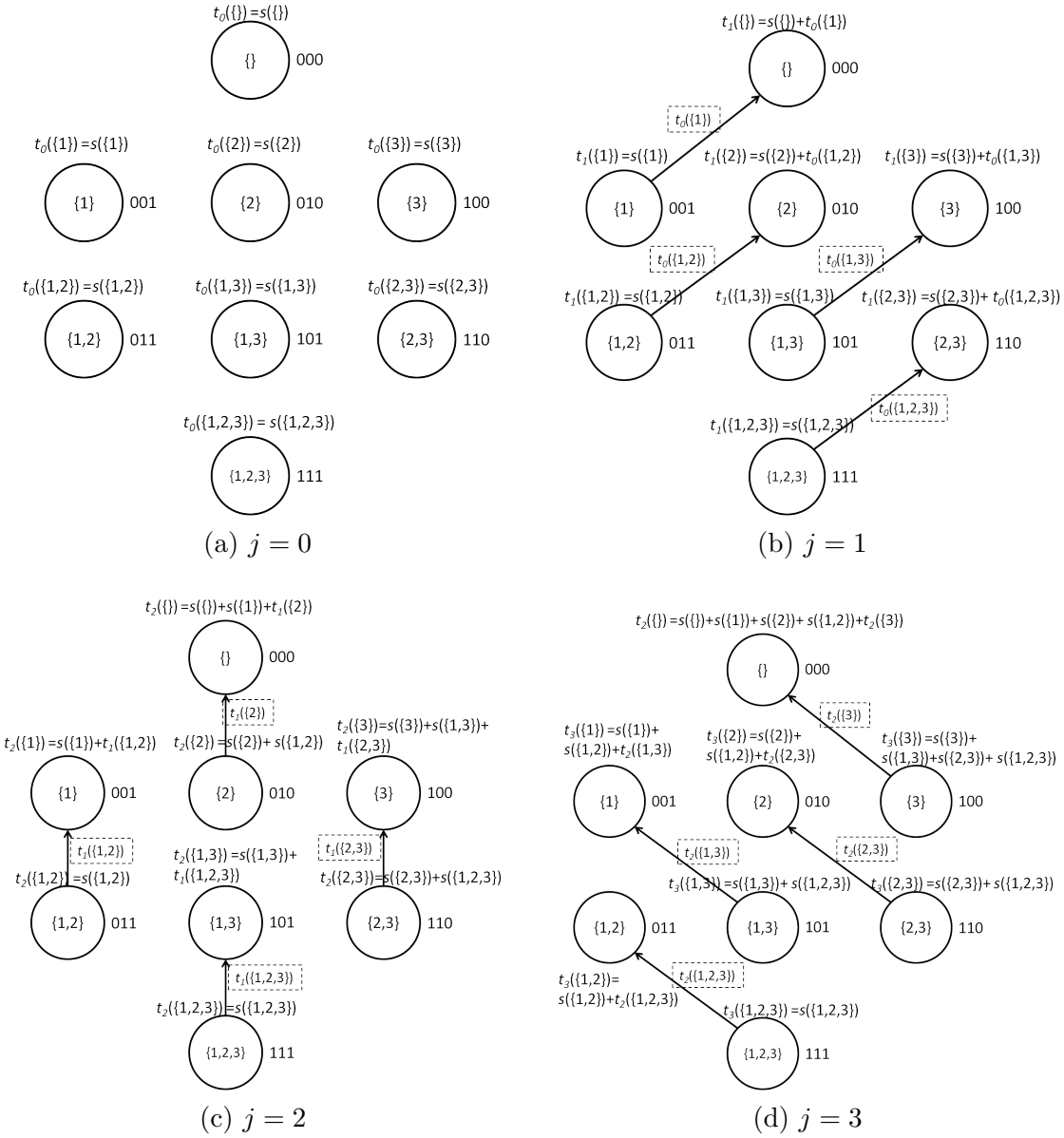


Figure 4.4: An illustrative example of parallel truncated downward zeta transform on n-D hypercube. In this case, $n = 3$, $d = 2$. The algorithm takes four iterations. Iterations 0, 1, 2 and 3 are show in (a), (b), (c) and (d), respectively. The functions in dashed boxes are the messages sending between the processors. In each iteration, the computed $t_j(S)$ is shown above each processor.

1, i.e., $\omega' = \omega \oplus 2^{j-1}$ (Line 3).[5] The neighbor receiving this t_{j-1} will perform the addition on line 10 in iteration j if necessary. Figure 4.3 illustrates an example of Algorithm 4.2 solving a problem with $n = 3$ and $d = 2$.

In Algorithm 4.3, the mapping of the computation of S to n-D hypercube is the same as in Algorithm 4.2. In iteration j, after the computation starts, each processor ω with $\omega[j] = 1$ sends its t_{j-1} to its neighbor ω' obtained by inverting its $\omega[j]$ to 0, i.e., $\omega' = \omega \oplus 2^{j-1}$. The neighbor receiving this t_{j-1} will perform the addition on line 7 in iteration j if necessary. Figure 4.4 illustrates an example of Algorithm 4.3 solving a problem with $n = 3$ and $d = 2$.

In each iteration, all $S \subseteq V$ are computed concurrently on a n-D hypercube, the computation times for both parallel algorithms being $O(n)$. Further, in both algorithms, the communications happen only between neighboring processors (two binary strings ω, ω' differ in only one bit). Thus, the two algorithms are communication-efficient.

We can use Algorithm 4.2 to compute $A_i(S)$ for a given $i \in V$ and all $S \subseteq V - \{i\}$ by setting $s(\cdot) = B_i(\cdot)$ and then computing $A_i(S) = q_i(S) \cdot t(S)$.[6] Note that $B_i(S)$ for any $S \subseteq V - \{i\}$ and $|S| \leq d$ has also been computed on processor corresponding to S since $s(S) = B_i(S)$. To compute $A_i(S)$ for all $i \in V$, we run Algorithm 4.2 n times with each time switching to the corresponding q_i and B_i functions. Thus, A_i for all $i \in V$ can be computed in $|V| \cdot O(n) = O(n^2)$ time. Each processor ω computes and keeps the corresponding $A_i(S)$ for all $i \in V$, which is the assumption we made in section 4.3.1.1. Thus, the mapping adopted by the two algorithms is well suited for our algorithm as it avoids a large number of messages to be passed when the computation transits to the next step.

We will use Algorithm 4.3 to compute Γ_v for a given $v \in V$. However, before applying the algorithm, we shall first compute $q_v(S)F(S)R(V - \{v\} - S)$ on the processor corresponding to S (see Equation 4.11). However, $F(S)$ and $R(V - \{v\} - S)$ are not on the same processor at the time when we have computed functions F and R. Fortunately, they are on the processors who are neighbors in the hypercube. Thus, the processor ω who has $F(S)$ shall retrieve $R(V -$

[5] $\oplus$ stands for the bitwise exclusive or (XOR) between two binary strings. 2^{j-1} stands for the binary string of integer 2^{j-1}.

[6] $A_i(S)$ are defined for all $S \subseteq V - \{i\}$, instead of $S \subseteq V$. However, the algorithm can still be deployed by setting $t(S) = 0$ for all $S \subseteq V$ s.t. $i \in S$.

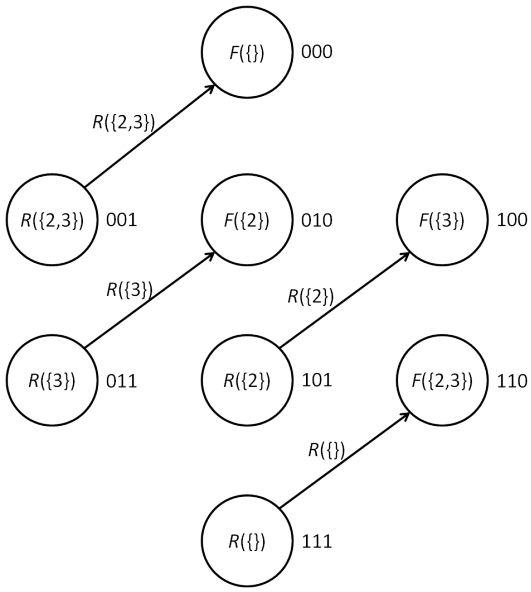


Figure 4.5: Retrieve R for computing Γ_v. The example shows the case in which Γ_v for $v = 1$ is computed.

$\{v\} - S$) from its neighbor ω' obtained by inverting its $\omega[v]$ to 1, i.e., $\omega' = \omega \oplus 2^{v-1}$ (see example in Figure 4.5), and compute $q_v(S)F(S)R(V - \{v\} - S)$ before Algorithm 4.3 is run. Then with Algorithm 4.3, computing Γ_v for any fixed $v \in V$ takes $O(n)$. The time for computing $\Gamma_v(G_v)$ for all $v \in V$ and $G_v \subseteq V - \{v\}$ and $|G_v| \leq d$ is therefore $|V| \cdot O(n) = O(n^2)$.

4.3.1.3 Computing $P(u \to v|D)$

With $B_v(G_v)$ and $\Gamma_v(G_v)$ computed, we can compute $P(u \to v, D)$ using Equation 4.10. Noting that $B_v(G_v)$ and $\Gamma_v(G_v)$ for any $G_v \subseteq V - \{v\}$ are on the same processor, each processor first computes $B_v(G_v) \cdot \Gamma_v(G_v)$ locally, then a `MPI_Reduce`, a collective function in MPI library is executed on the hypercube to compute the sum of $B_v(G_v) \cdot \Gamma_v(G_v)$ from all processors. $P(u \to v|D)$ is then obtained by evaluating $P(u \to v, D)/F(V)$ at the processor with the highest rank, i.e., all bits in its *id* are 1's. A `MPI_Reduce` operation on a n-D hypercube requires $O((\tau + \mu m)n)$ time, where τ, μ, m are constants, specifying the latency, bandwidth of the communication network, and the message size. Thus, computing $P(u \to v|D)$ for all $u, v \in V, u \neq v$ takes $O((\tau + \mu m)n^3)$ time.

Adding up the time for each step, the time for evaluating all $n(n-1)$ edges is $O(n^3)$. As the sequential run-time is $O(2n(d+1)2^n)$, the parallel efficiency is $\Theta(2(d+1)/n^2)$.

4.3.2 k-D Hypercube Algorithm

In section 4.3.1, we have described the development of our parallel algorithm on an n-D hypercube. However, we usually expect the number of processors $p \ll 2^n$. Let $p = 2^k$ be the number of processors, where $k < n$. We assume that the processors can communicate as in a k-D hypercube. The strategy is to decompose the n-D lattice into 2^{n-k} k-D lattices and map each k-D lattice to the $p = 2^k$ processors (k-D hypercube).

Following our previous definition, we use the binary string ω to denote the corresponding hypercube node S. We number the positions of a binary string using $1, ..., n$ (from right-most bit to left-most bit), and use $\omega[i, j]$ to denote the substring of ω between and including positions i and j. We partition the n-D lattice into 2^{n-k} k-D lattices based on the left $n-k$ bits of node id's. For a lattice node ω, $\omega[k+1, n]$ specifies the k-D lattice it is part of and $\omega[1, k]$ specifies the id of the processor it is assigned to. As an example, Figure 4.6 shows the decomposition of an 3-D lattice to two 2-D lattices and the mapping to an 2-D hypercube computing cluster. In this case, subsets $\{\}$ and $\{3\}$ are assigned to processor `00`, $\{1\}$ and $\{1, 3\}$ are assigned to processor `01`, so on and so forth. Thus, each processor in a k-D hypercube is responsible for computing relevant scores for 2^{n-k} S subsets. This forms the basic idea of our k-D hypercube algorithm.

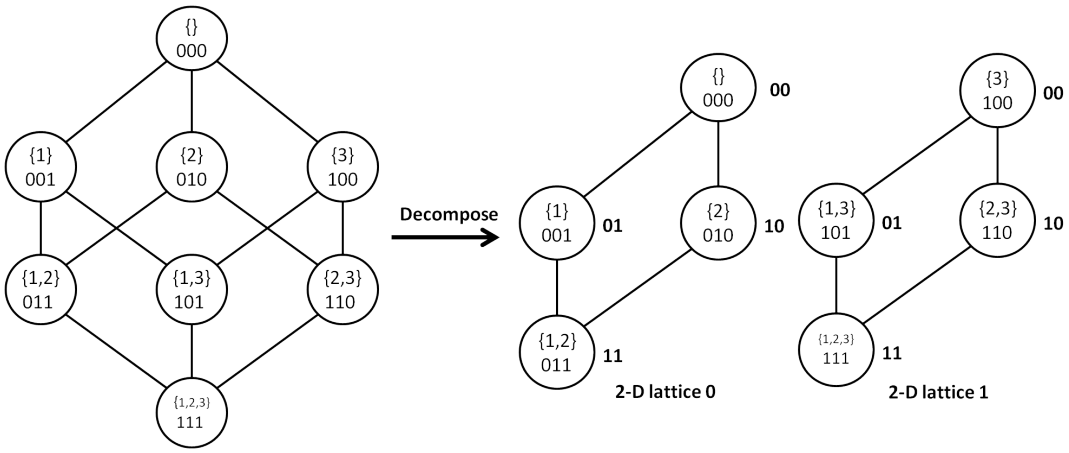


Figure 4.6: Decompose a 3-D lattice into two 2-D lattices which are then mapped to an 2-D hypercube. The 3-bit binary string inside the node represents the binary code of the corresponding subset S. The 2-bit binary string beside each node denotes the id of the processor in the 2-D hypercube.

In the following, we first develop k-D hypercube algorithms for the two *zeta* transform

variants. We then present the k-D hypercube algorithms for computing F and R functions. Finally we introduce the overall k-D hypercube algorithm for computing the edge posteriors.

4.3.2.1 Parallel Fast Zeta Transforms on k-D hypercube

In order to compute A and Γ functions on a k-D hypercube, we generalize Algorithms 4.2 and 4.3. We number the processors in the k-D hypercube computer cluster with a k-bit binary string r such that two adjacent processors r, r' differ in one bit. The basic idea is, instead of computing the transform for only one subset S, each processor r is responsible for computing 2^{n-k} subsets S such that $r = \omega[1,k]$. We present the generalized algorithms for the two transforms in Algorithm 4.4 and Algorithm 4.5, respectively.

Algorithm 4.4 Parallel Truncated Upward Zeta Transform on k-D hypercube

Assumption: $1 \leq k \leq n$, each subset $S \subseteq V$ is encoded by an n-bit binary string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Each processor in the k-D hypercube is encoded by an k-bit binary string r. Subset S is computed on processor $r = \omega[1,k]$.

On each processor r, $t_0(S) \leftarrow s(S)$ for all $|S| \leq d$ s.t. $r = \omega[1,k]$ and $t_0(S) \leftarrow 0$ otherwise.
for $j \leftarrow 1$ to n **do**
  for each $S \subseteq V$ with $|S \cap \{j+1, ..., n\}| \leq d$ on each processor r **do**
    $t_j(S) \leftarrow 0$
    if $|S \cap \{j, ..., n\}| \leq d$ **then**
      $t_j(S) \leftarrow t_{j-1}(S)$
    end if
    if $j \in S$ **then**
      if $j \leq k$ **then**
        Retrieve $t_{j-1}(S - \{j\})$ from processor $r' = r \oplus 2^{j-1}$.
      end if
      $t_j(S) \leftarrow t_j(S) + t_{j-1}(S - \{j\})$
    end if
  end for
end for
return $t_n(S)$

Figure 4.7 shows a running example of Algorithm 4.4 with $n = 3$, $d = 2$ and $k = 2$. In this case, we have 8 subsets and each processor is computing two subsets. Another notable difference from the example in Figure 4.3 is that in j-th iteration where $j > k$, $t_{j-1}(S-\{j\})$'s are available locally thus no message passing between processor is required. Similarly, Figure 4.8 shows a running example of Algorithm 4.5 with $n = 3$, $d = 2$ and $k = 2$.

We now present two theorems that respectively characterize the run-time complexities of the two algorithms.

Algorithm 4.5 Parallel Truncated Downward Zeta Transform on k-D hypercube

Assumption: $1 \leq k \leq n$, each subset $S \subseteq V$ is encoded by an n-bit binary string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Each processor in the k-D hypercube is encoded by an k-bit binary string r. Subset S is computed on processor $r = \omega[1, k]$.

1: On each processor, $t_0(S) \leftarrow s(S)$ for all S s.t. $r = \omega[1, k]$.
2: **for** $j \leftarrow 1$ to n **do**
3: **for** each $S \subseteq V$ with $|S \cap \{1, ..., j\}| \leq d$ on each processor r **do**
4: $t_j(S) \leftarrow t_{j-1}(S)$
5: **if** $j \notin S$ **then**
6: **if** $j \leq k$ **then**
7: Retrieve $t_{j-1}(S \cup \{j\})$ from processor $r' = r \oplus 2^{j-1}$.
8: **end if**
9: $t_j(S) \leftarrow t_j(S) + t_{j-1}(S \cup \{j\})$
10: **end if**
11: **end for**
12: **end for**
13: **return** $t_n(S)$

Theorem 4.1. *Algorithm 4.4 computes the truncated upward zeta transform in time* $O((d+1) \cdot 2^{n-k} + k(n-k)^d)$ *on* k-D *hypercube.*

Proof. As it is specified, each processor r computes subsets S s.t. $r = \omega[1, k]$. Algorithm 4.4 runs for n iterations. For the iterations $j = n - d, ..., n$, all $S \subseteq V$ satisfy the condition on line 3, thus each processor performs the computation on lines 4–13 for the corresponding 2^{n-k} subsets on it. The total computing time for these iterations is $O((d+1)2^{n-k}) = O(d2^{n-k})$.

For iterations $j = 1, ..., n - d - 1$, the processor r s.t. $r[i] = 0$ for all $i = 1, .., k$ has the largest number of subset S that satisfy the condition on line 3, thus computes lines 4–13 the most frequently among all the processors. The computation time of the algorithm for these iterations is up-bounded by its computation time. Thus, for iterations $j = 1, ..., n - d - 1$, we only need to characterize this processor's computation time, which is proportional to

$$\begin{aligned}
&\sum_{j=1}^{k}\sum_{r=0}^{d}\binom{n-k}{r} + \sum_{j=k+1}^{n-d-1} 2^{j-k}\sum_{r=0}^{d}\binom{n-j}{r} = k\sum_{r=0}^{d}\binom{n-k}{r} + 2^{-k}\sum_{j=k+1}^{n-d-1} 2^j \sum_{r=0}^{d}\binom{n-j}{r} \\
&\leq k\sum_{r=0}^{d}\binom{n-k}{r} + 2^{-k}\sum_{j=k+1}^{n-d-1} 2^j (n-j)^d \leq k\sum_{r=0}^{d}\binom{n-k}{r} + 2^{-k}\sum_{j=1}^{n-d-1} 2^j (n-j)^d \\
&\leq k\sum_{r=0}^{d}\binom{n-k}{r} + 2^{-k}2^n\sum_{j=0}^{\infty}(1/2)^j j^d
\end{aligned} \tag{4.12}$$

The first term $k\sum_{r=0}^{d}\binom{n-k}{r} = O(k(n-k)^d)$. The second term $2^{-k}2^n\sum_{j=0}^{\infty}(1/2)^j j^d = O(2^{n-k})$ as the infinite sum converges to a finite limit for a fixed d. Thus, the time combined for all iteration is $O(k(n-k)^d) + O(2^{n-k}) + O(d2^{n-k}) = O((d+1)\cdot 2^{n-k} + k(n-k)^d)$. □

Theorem 4.2. *Algorithm 4.5 computes the truncated downward zeta transform in time* $O((4d+4)\cdot 2^{n-k})$ *on k-D hypercube.*

Proof. Each processor r computes subsets S s.t. $r = \omega[1,k]$. In Algorithm 4.5, line 1 takes $O(2^{n-k})$ time. Lines 2–12 runs for n iterations. For the iterations $j = 1,...,d$, all $S \subseteq V$ satisfy the condition on line 3, thus each processor performs the computation on line 4–10 for all 2^{n-k} subsets on it. Thus the total computation time for these iterations is $O(d2^{n-k})$.

For iterations $j = d+1,...,n$, the processor r s.t. $r[i] = 0$ for all $i = 1,..,k$ enters the loop 3–11 more frequently than any other processors, thus requires the most computation time. The running time of Algorithm 4.5 for these iterations is up-bounded by its running time, which is proportional to

$$
\begin{aligned}
&\sum_{j=d+1}^{k+d} 2^{n-k} + \sum_{j=k+d+1}^{n} 2^{n-j}\sum_{r=0}^{d}\binom{j-k}{r} = k2^{n-k} + 2^{-k}\sum_{i=d+1}^{n-k} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} \\
&= k2^{n-k} + 2^{-k}\sum_{i=d+1}^{4d-1} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} + 2^{-k}\sum_{i=4d}^{n} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} - 2^{-k}\sum_{i=n-k+1}^{n} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} \\
&\le k2^{n-k} + 2^{-k}\sum_{i=d+1}^{4d-1} 2^{n} + 2^{-k}\sum_{i=4d}^{n-k} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} - 2^{-k}\sum_{i=n-k+1}^{n} 2^{n-d}\sum_{r=0}^{d}\binom{d}{r} \\
&= k2^{n-k} + (3d-1)2^{n-k} + 2^{-k}\sum_{i=4d}^{n-k} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} - 2^{-k}\sum_{i=n-k+1}^{n} 2^{n-d}2^{d} \\
&= (k+3d-1)2^{n-k} + 2^{-k}\sum_{i=4d}^{n} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} - k2^{n-k} \\
&\le (3d-1)2^{n-k} + 5\cdot 2^{n-k}
\end{aligned}
\tag{4.13}
$$

The upper bound $2^{-k}\sum_{i=4d}^{n} 2^{n-i}\sum_{r=0}^{d}\binom{i}{r} \le 5\cdot 2^{n-k}$ in last step is from **Corollary 3** in (Koivisto, 2006a). Thus, the run-time is $O((4d+4)\cdot 2^{n-k})$. □

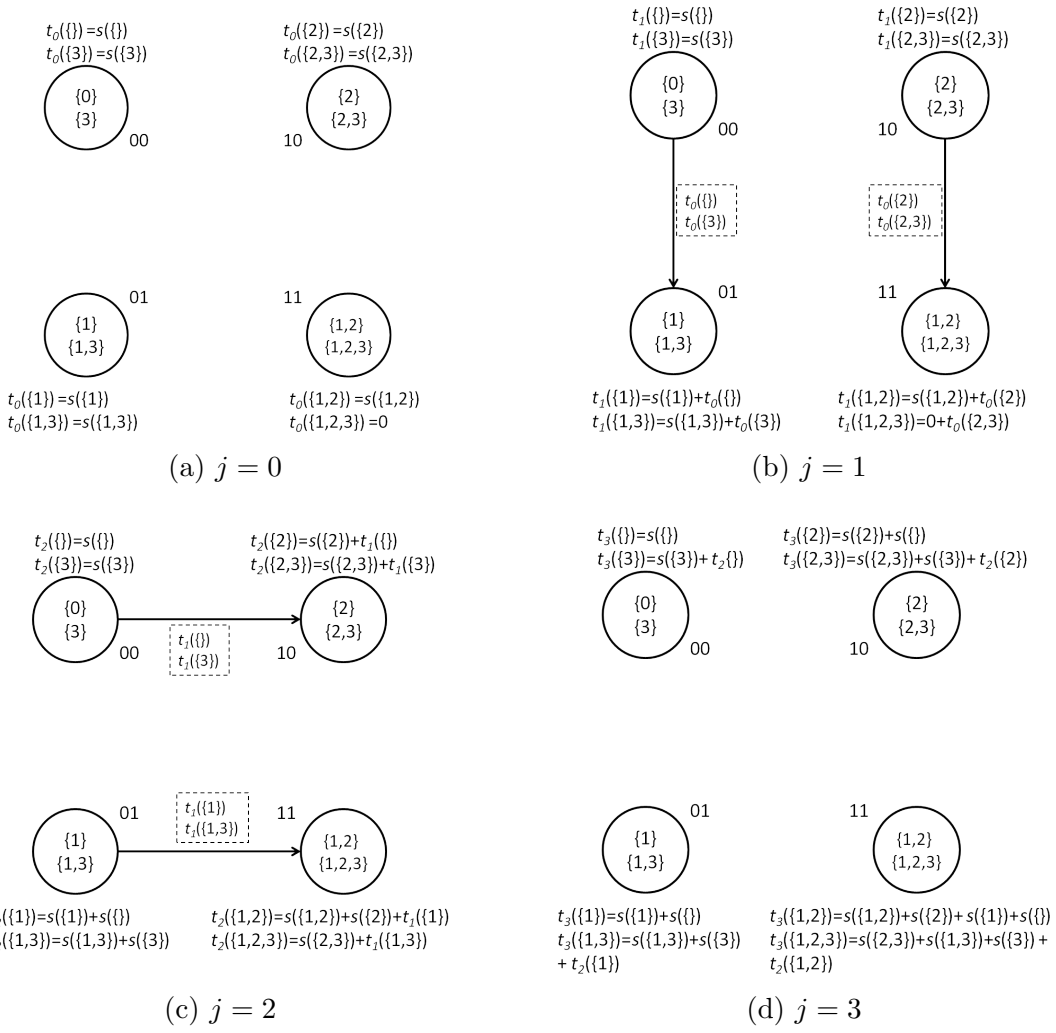


Figure 4.7: An illustrative example of parallel truncated upward zeta transform on k-D hypercube. In this case, $n=3$, $d=2$, $k=2$. The algorithm takes four iterations. Iterations 0, 1, 2 and 3 are show in (a), (b), (c) and (d), respectively. The functions in dashed boxes are the messages sending between the processors. In each iteration, the computed $t_j(S)$'s is shown under each processor. In (d), $j=3>k=2$, $t_{j-1}(S-\{j\})$'s are available locally thus no message passing is needed.

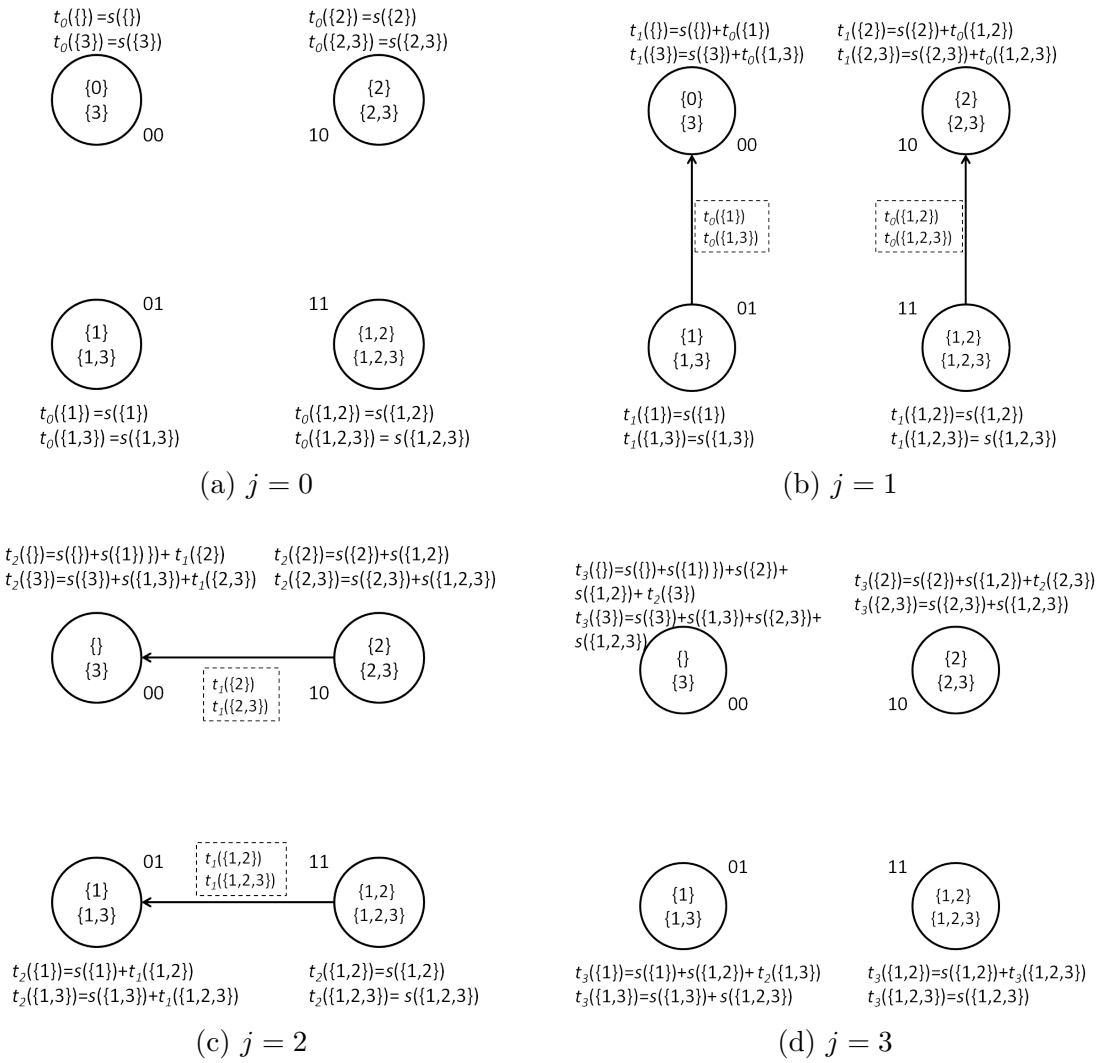


Figure 4.8: An illustrative example of parallel truncated downward zeta transform on k-D hypercube. In this case, $n=3$, $d=2$, $k=2$. The algorithm takes four iterations. Iterations 0, 1, 2 and 3 are show in (a), (b), (c) and (d), respectively. The functions in dashed boxes are the messages sending between the processors. In each iteration, the computed $t_j(S)$'s is shown under each processor. In (d), $j=3>k=2$, $t_{j-1}(S\cup\{j\})$'s are available locally thus no message passing is needed.

4.3.2.2 Computing $F(S)$ and $R(S)$ on k-D Hypercube

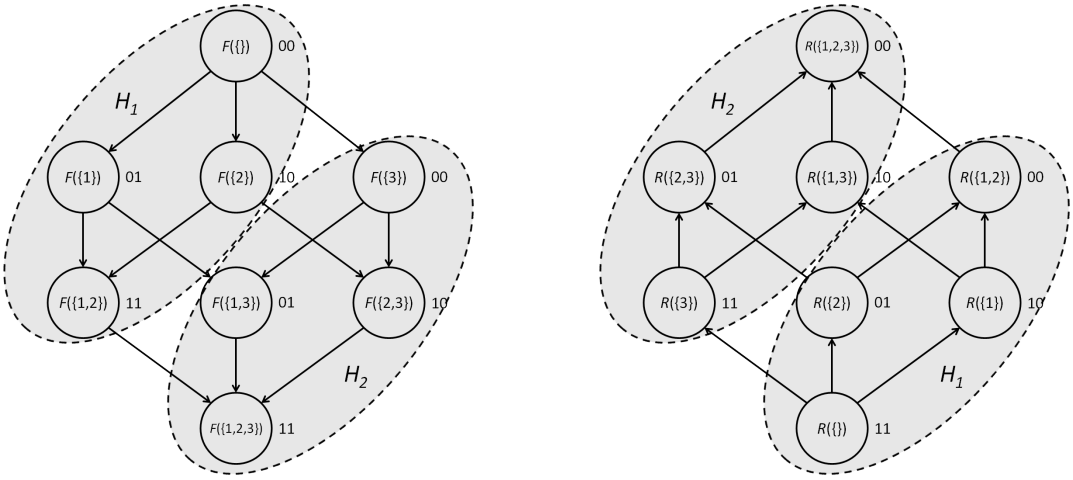
To compute function F, we partition the n-D DP lattice into 2^{n-k} k-D hypercubes based on the left $n-k$ bits of node id's. For a lattice node ω, $\omega[k+1,n]$ specifies the k-D hypercube it is part of and $\omega[1,k]$ specifies the processor it is assigned to. Using the strategy proposed by Nikolova et al. (2009), we pipeline the execution of the k-D hypercubes to complete the parallel execution in $2^{n-k}+k$ time steps such that all processors are active except for the first k and last k time steps during the buildup and finishing off of the pipeline. Specifically, let each k-D hypercube denoted by an $(n-k)$ bit string, which is the common prefix to the 2^k lattice/k-D hypercube nodes that are part of this k-D sub-hypercube. The k-D hypercubes are processed in the increasing order of the number of 1's in their bit string specifications, and in lexicographic order within the group of hypercubes with the same number of 1's. Formally, we have the following rule: let H_i and H_j be two k-D hypercubes and let ω_S and ω_T be the binary strings of two nodes S and T in the lattice that map to H_i and H_j, respectively. Then, the computation of H_i is initiated before computation of H_j if and only if:

1. $\mu(\omega_S[k+1,n]) < \mu(\omega_T[k+1,n])$, or
2. $\mu(\omega_S[k+1,n]) = \mu(\omega_T[k+1,n])$ and $\omega_S[k+1,n]$ is lexicographically smaller than $\omega_T[k+1,n]$.

Figure 4.9a illustrates a case of computing $F(S)$ with $n=3$ and $k=2$. In this example, the 3-D F lattice is partitioned to two 2-D hypercubes H_1 and H_2. H_1 is processed before H_2 is processed. One feature of the pipelining is that once a processor completes its computation in one k-D hypercube, it transits to next k-D hypercube immediately without waiting for other processors to complete their computations in current hypercube. In Figure 4.9a, for example, once the processor 00 completes node $\{\}$ and sends out data, it starts on node $\{3\}$ even if processors 01, 10, 01 are still working on their nodes in H_1. This feature prevents processors from excessive idling during the transitions between consecutive hypercubes.

The strategy to compute function $R(S)$ is similar. The only difference is the mapping of the subsets to processors. $R(S)$ is assigned to the processor with id $r = \neg\omega[1,k]$ [7], i.e., $R(S)$ is

[7] $\neg\omega[1,k]$ denotes the bitwise complement of binary string $\omega[1,k]$.



(a) Computing $F(S)$ (b) Computing $R(S)$

Figure 4.9: Pipelining execution of hypercubes to compute $F(S)$ and $R(S)$. The example shows a case with $n = 3$ and $k = 2$.

computed on the processor where $F(V - S)$ is computed. In other words, processors operate in the reverse order as that when they compute $F(S)$. An example of computing a 3-D R lattice on 2-D hypercube is shown in Figure 4.9b.

4.3.2.3 Overall Algorithm: ParaREBEL

With the k-D algorithms for the two transforms, A_i (and B_i) and Γ_v functions can be computed efficiently. As mentioned, each processor with *id* r is responsible for computing 2^{n-k} subsets S such that $r = \omega[1, k]$. Note that before computing Γ_v, we need compute $q_v(S)F(S)R(V - \{v\} - S)$, where $F(S)$ and $R(V - \{v\} - S)$ are not necessarily on the same processor in the k-D hypercube. Fortunately, with our partition strategy, $F(S)$ and $R(V - \{v\} - S)$ locate either on the same processor or on the neighboring processors in the k-D hypercube. Specifically, when $v \leq k$, processor r with $r[v] = 0$ need retrieve $R(S)$ from its neighbor $r' = r \oplus 2^{v-1}$; when $v > k$, $F(S)$ and $R(V - \{v\} - S)$ are on the same processor thus no message passing is needed to compute $q_v(S)F(S)R(V - \{v\} - S)$.

Finally, to compute $P(u \to v, D)$ for any $u, v \in S, u \neq v$, each processor r first adds up all local $B_v(G_v)\Gamma_v(G_v)$ scores with $\omega_{G_v}[1, k] = r$, then a `MPI_Reduce` is launched on the k-D

hypercube to obtain the sum. The posteriors $P(u \to v|D)$ are evaluated as $P(u \to v, D)/F(V)$ on the processor r with $r[i] = 1$ for all $i \in \{1, ..., k\}$.

Algorithm 4.6 ParaREBEL computes the posterior probabilities of all $n(n-1)$ edges with $p = 2^k$ processors.

Assumption: each subset $S \subseteq V$ is encoded by an n-bit string ω, where $\omega[i] = 1$ if variable $i \in S$ and $\omega[i] = 0$ otherwise. Each processor in the k-D hypercube is encoded by an k-bit string r.

1: **for** each $i \in V$, compute $B_i(S)$ and $A_i(S)$ for all $S \subseteq V - \{i\}$ by Algorithm 4.4. Each processor r computes subsets S s.t. $r = \omega[1, k]$.
2: Compute $F(S)$ for all $S \subseteq V$ on k-D hypercube. Each processor r computes subsets S s.t. $r = \omega[1, k]$.
3: Compute $R(S)$ for all $S \subseteq V$ on k-D hypercube. Each processor r computes subsets S s.t. $r = \neg\omega[1, k]$.
4: **for** each $v \in V$ **do**
5: **if** $v \leq k$ **then**
6: Each processor r with $r[v] = 1$ sends all its R scores to its neighbor $r' = r \oplus 2^{v-1}$.
7: **end if**
8: Each processor r with $r[v] = 0$ computes $q_v(S)F(S)R(V - \{v\} - S)$ for all its S.
9: Compute $\Gamma_v(G_v)$ for all $G_v \subseteq V - \{v\}$ with $|G_v| \leq d$ by Algorithm 4.5.
10: **for** each $u \in V - \{v\}$ **do**
11: Each processor r recomputes $B_v(G_v)$ for all G_v with $r = \omega_{G_v}[1, k]$, then adds up all local $B_v(G_v)\Gamma_v(G_v)$ scores with $|G_v| \leq d$.
12: `MPI_Reduce` is executed on the k-D hypercube to compute the sum of all $B_v(G_v)\Gamma_v(G_v)$, $P(u \to v, D)$, obtained on processor r with $r[i] = 1$ for all $i \in \{1, ..., k\}$.
13: Processor r with $r[i] = 1$ for all $i \in \{1, ..., k\}$ evaluates $P(u \to v|D) = P(u \to v, D)/F(V)$.
14: **end for**
15: **end for**

The overall k-D hypercube algorithm, named as ParaREBEL[8] (*Parallel Rapid Exact Bayesian Edge Learning*), is outlined in Algorithm 4.6.

4.3.2.4 Time and Space Complexity

We characterize the running time of *ParaREBEL* under the assumption that the maximum in-degree d is a constant.

For any fixed $i \in V$, computing $A_i(L_i)$ for all $L_i \subseteq V - \{i\}$ takes $O((d+1) \cdot 2^{n-k} + k(n-k)^d)$ time (Theorem 4.1). Thus, line 1 takes $|V| \cdot O((d+1) \cdot 2^{n-k} + k(n-k)^d) = O((d+1)n2^{n-k} + kn(n-k)^d)$ time to compute B_i and A_i scores for all $i \in V$.

Line 2 and line 3 take $O(n(2^{n-k} + k))$ time each as we pipeline the execution of the k-D hypercubes in $2^{n-k} + k$ steps and each step costs $O(n)$.

In line 9, for any $v \in V$, computing Γ_v scores takes $O((4d + 4) \cdot 2^{n-k})$ time (Theorem 4.2).

[8] The serial algorithm in (Koivisto, 2006a) is called REBEL.

Line 11 takes $O(\frac{n^d}{2^k})$ time as there are no more than $O(\frac{n^d}{2^k})$ $B_v(S)\Gamma_v(S)$ scores on each processor if bounded in-degree d is assumed. In line 12, `MPI_Reduce` procedure takes $O((\tau+\mu m)k)$ time. Thus, the time combined for Lines 4-15 is $O((((\tau+\mu m)k+\frac{n^d}{2^k})n+(4d+4)2^{n-k})\cdot n) = O(kn^2+\frac{n^{d+2}}{2^k}+4(d+1)n2^{n-k}) = O(kn^2+4(d+1)n2^{n-k})$.[9]

The total time for the overall algorithm is therefore $O(5(d+1)n2^{n-k}+kn(n-k)^d+kn^2) = O(5(d+1)n2^{n-k}+kn(n-k)^d)$.[10].

Furthermore, B, A, Γ, F, R scores are evenly distributed on the 2^k processors. Therefore, the storage per processor used by the parallel algorithm is $O(n2^{n-k})$. Since the space requirement of the sequential algorithm is $O(n2^n)$, our parallel algorithm achieves the optimal space efficiency.

In summary, we obtain the following results.

Theorem 4.3. *Algorithm ParaREBEL runs in time* $O(5(d+1)n2^{n-k}+kn(n-k)^d)$ *and space* $O(n2^{n-k})$ *per processor.*

4.4 Experiments

In this section, we present the experiments for evaluating our ParaREBEL algorithm.

4.4.1 Implementation and Computing Environment

We implemented the proposed ParaREBEL algorithm[11] in C++ and MPI and demonstrated its scalability on TACC Stampede[12], a Dell PowerEdge C8220 cluster. Each computing node in the cluster consists of two Xeon Intel 8-Core E5-2680 processors (16 cores in all), sharing 32 GB memory. All experiments were run with one MPI process per core. To allow more memory per process, only 8 cores in each node were recruited so that each process could use up to 4 GB memory. The maximum number of nodes/cores allowed for a regular user on TACC Stampede

[9] $O(kn^2+\frac{n^{d+2}}{2^k}+n2^{n-k}) = O(kn^2+n2^{n-k})$ because $\frac{n^{d+2}}{2^k}$ is dominated by $n2^{n-k}$.

[10] We normally have $d \geq 2$, i.e., the up-bound of the in-degree is at least 2. In this case, $kn(n-k)^d$ dominates kn^2.

[11] ParaREBEL is available for download at http://www.cs.iastate.edu/~yetianc/software.html.

[12] http://www.tacc.utexas.edu/resources/hpc/stampede

is 256/4096. To maintain 4 GB per core, we can only use up to 2048 cores. Thus, all the following experiments were done on up to 2048 cores.

4.4.2 Running Time and Memory Usage

We first evaluated the time and space complexity of our algorithm. We compared our implementation with REBEL[13], a C++ implementation of the serial algorithm (Algorithm 4.1) in (Koivisto, 2006a).

We generated a set of synthetic data sets with discrete random variables. Each dataset contains 500 samples. For each data set, we ran the serial algorithm and our ParaREBEL algorithm to compute the posterior probabilities for all $n(n-1)$ potential edges. We did two tests: one with varying bounded in-degree d and fixed number of variables n, the other with varying number of variables n and fixed bounded in-degree d. In both tests, the total running times were recorded and *speedup* and *efficiency* were computed. In the second test, the memory usages per processor were collected and the total memory usages were calculated.

In the first test, we fixed $n = 25$ and studied the performance of ParaREBEL algorithm with respect to the bounded in-degree d ($d = 2, 4, 6, 8$). The run-times are presented in Table 4.1. The corresponding speedups and efficiencies are illustrated in Figure 4.10. Generally, we observed overall good scaling (see speedup plot in Figure 4.10) for all values of d. The speedup and efficiency both improve when d increases from 2 to 4, but decline when d keeps increasing from 4 through 6 to 8. From our theoretical analysis of running time , we have $speedup = \frac{2(d+1)n2^n}{5(d+1)n2^{n-k}+kn(n-k)^d} = \frac{2\cdot 2^n}{5\cdot 2^{n-k}+\frac{k(n-k)^d}{d+1}}$ and $efficiency = \frac{2(d+1)n2^n}{5(d+1)n2^n+kn(n-k)^d 2^k} = \frac{2\cdot 2^n}{5\cdot 2^n+\frac{k(n-k)^d}{d+1}2^k}$. Both formulas are not a monotonic function of d. when d is small, $d+1$ in the denominator of $\frac{k(n-k)^d}{d+1}$ dominates thus both speedup and efficiency improve when d increases. When d is large, $k(n-k)^d$ starts to dominate and the two measures decline when d increases. Thus, our empirical result is consistent with our theoretical result. For $d = 4$, the efficiencies are maintained above 0.53 with up to 2048 cores.[14]

In the second test, we fixed $d = 4$ and studied the performance of the algorithm with respect

[13] http://www.cs.helsinki.fi/u/mkhkoivi/REBEL

[14] Generally, parallel algorithms with *efficiency*≥ 0.5 are considered to be successfully parallelized.

Table 4.1: Run-time for the test data with $n = 25$ with varying bounded in-degree d.

No.CPUs	**Run-time (seconds)**			
	$d = 2$	$d = 4$	$d = 6$	$d = 8$
Serial	1319	2295	4308	7739
4	1284	1330	1500	2383
8	575	594	711	1304
16	327	338	417	764
32	139	146	181	466
64	59.9	64.2	102	268
128	26.6	29.4	55.6	153
256	11.7	13.8	31.3	86.8
512	5.2	6.8	18.2	48.5
1024	2.5	3.6	11.0	26.9
2048	1.5	2.1	6.7	14.8

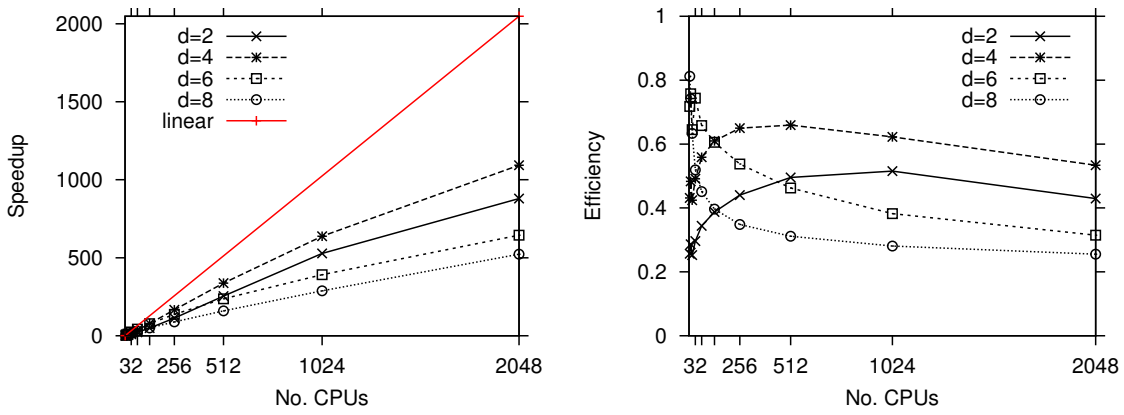


Figure 4.10: *Speedup* and *efficiency* for the test data set with $n = 25$ with varying bounded in-degree d. The red diagonal line in speedup plot represents the linear or ideal speedup, i.e., the up-bound that a parallel algorithm can achieve in theory.

to the number of variables ($n = 21, 23, 25, 27, 29, 31, 33$). We first compared the run-times. As showed in Table 4.2, the run-times are reflective of the exponential dependence on n. Further, we observed that the algorithm scaled much better when n becomes larger (see speedup and efficiency plot in Figure 4.11). This is also supported by our theoretical result. With a minor transform, our running time analysis suggests $speedup = \frac{2(d+1)}{5(d+1)2^{-k}+k(n-k)^d 2^{-n}}$. When n is large enough, speedup (and efficiency) is a increasing function of n. For $n = 25$, the parallel algorithm maintains an efficiency of about 0.6 with up to 2048 cores. For $n = 33$, the problem can only be solved on 1024 and 2048 cores due to memory constraint. We had a try on $n = 34$ using 2048 cores but were not able to solve it as it ran out of memory.

Table 4.2: Run-time for the test data sets with $n = 21, 23, 25, 27, 29, 31, 33$ with fixed $d = 4$.

No.CPUs	**Run-time (seconds)**						
	$n = 21$	$n = 23$	$n = 25$	$n = 27$	$n = 29$	$n = 31$	$n = 33$
Serial	96.5	492	2295	-	-	-	-
4	44.1	252	1330	-	-	-	-
8	17.2	94.2	594	-	-	-	-
16	10.3	55.5	338	-	-	-	-
32	5.0	25.5	146	682	-	-	-
64	2.7	11.9	64.2	385	2201	-	-
128	1.6	5.8	29.4	167	864	-	
256	0.97	3.2	13.8	73.5	389	2540	-
512	0.61	1.8	6.8	33.9	196	987	-
1024	0.4	1.1	3.6	15.9	87	488	2884
2048	0.27	0.7	2.1	7.8	39	215	1452

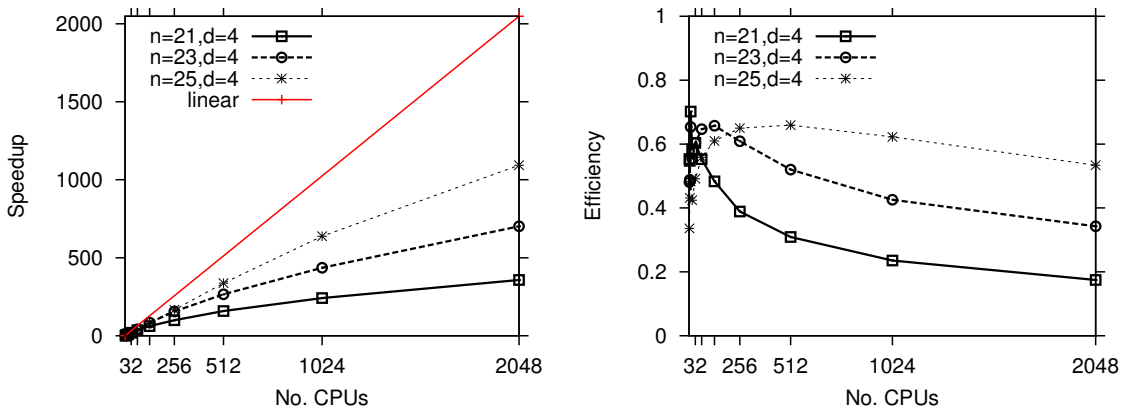


Figure 4.11: *Speedup* and *efficiency* for the test data sets with $n = 21, 23, 25$. The red diagonal line in speedup plot represents the linear or ideal speedup, i.e., the up-bound that a parallel algorithm can achieve in theory.

One interesting observation is that for any fixed d and n, the parallel efficiency increases as the No.CPUs increases, peaks at somewhere in between, then gradually decreases as No.CPUs goes up to 2048 CPUs (see efficiency plot in Figure 4.11). Mathematically, this optimum can be found by maximizing *efficiency*$= \frac{2(d+1)2^n}{5(d+1)2^n+k(n-k)^d 2^k}$, i.e., minimizing $k(n-k)^d 2^k$ over k. Solving this optimization problem yields $k^* = n(\ln 2 + 1)/(\ln 2 + 1 + d) \approx 1.7n/(d + 1.7)$. Plugging in $n = 25$ and $d = 4$ yields $k^* = 7.5 \approx 8$, i.e., $2^{k^*} \approx 256$ cores. Plugging in $n = 23$ and $d = 4$ yields $k^* = 6.8 \approx 7$, i.e., $2^{k^*} \approx 128$ cores. All these results consist exactly with the observation in Figure 4.11. This provides another piece of solid experimental evidence for Theorem 4.3. Further, this optimum k^* is proportional to n, i.e., the optimal *efficiency* will

Table 4.3: Memory usage for the test data with $n = 23, 25, 27, 29, 31, 33$ with fixed $d = 4$. The term outside the parentheses is the total memory usage measured in GB, the term inside the parentheses is the memory usage per core measured in MB. The missing entries indicate the cases where the program runs out of memory.

No.CPUs	**Memory Usage**					
	$n = 23$	$n = 25$	$n = 27$	$n = 29$	$n = 31$	$n = 33$
4	1.88 (481)	8.00 (2049)	-	-	-	-
8	1.88 (240)	8.00 (1025)	-	-	-	-
16	1.88 (121)	8.01 (513)	-	-	-	-
32	2.17 (70)	8.30 (266)	34.32 (1098)	-	-	-
64	2.49 (40)	8.62 (138)	34.64 (554)	144.68 (2315)	-	-
128	3.31 (27)	9.46 (76)	35.48 (284)	145.53 (1164)	-	-
256	4.93 (20)	11.06 (44)	37.09 (148)	147.07 (588)	606.13 (2425)	-
512	8.40 (17)	14.73 (30)	40.76 (82)	150.87 (302)	615.03 (1230)	-
1024	17.58 (18)	23.62 (24)	49.72 (50)	159.87 (160)	623.88 (624)	2520 (2520)
2048	41.08 (21)	47.32 (24)	72.97 (36)	183.69 (92)	647.27 (324)	2560 (1300)

be achieved by using larger number of cores when problem becomes larger. This suggests our ParaREBEL algorithm scales very well with respect to the problem size n.

We then examined the actual memory usages with respect to the number of variables n and the number of cores 2^k in Table 4.3. For $n = 23$, the total memory usage remains the same (1.88 GB) for $2^k = 4, 8, 16$ cores, but starts to increase as the number of cores increases from 16 to 2048. This increase is dramatic for the number of cores ranging from 256 to 2048, i.e, the memory usage is doubled when the number of cores is doubled. This can be explained by examining the memory usage per core. For $2^k = 4, 8, 16$, the memory usage per core decreases by half when the number of cores is doubled. This is consistent with our theoretical analysis that the space complexity is $O(n2^{n-k})$ per core. When $2^k \geq 16$, the reduction slows down and the memory usage plateaus at about 20 MB per core. It is speculated that in addition to the memory allocated for storing the B, A, F, R, Γ scores, each core requires extra $10 \sim 20$ MB memory to store program execution related data in order to run the program. This overhead is negligible when the memory usage per core is dominated by the scores but comes into play otherwise. For $n = 25$, total memory usage stays at about 8 GB for $2^k = 4 \sim 64$ and starts to increase thereafter; for $n = 27$, total memory usage stays at about 35 GB for $2^k = 32 \sim 256$ and starts to increase thereafter; for $n = 29, 31, 33$, the memory usage per core is dominated by the scores, thus, the total memory usage stays roughly constant with respect to the number

of cores examined. Further, it is easily observed that the memory usages (total memory usage and memory usage per core) are reflective of the exponential dependence on n. Thus, the observations on the memory usage are consistent with our analysis of the space complexity.

Moreover, the missing entries in the table are the cases where the program runs out of memory. Thus, we concluded that it requires at least 4 GB memory per core if $n - k > 23$. To solve a problem of $n \geq 34$, we need 2048 cores with more than 4 GB memory per core or 4096 cores with more than 2 GB memory per core. However, these resources are unavailable to a regular user on TACC Stampede. Further, we observed that the problem of $n = 33$ could be solved on 1024 cores in less than one hour, and 2048 cores in less than half an hour. The computation times are still far away from the practical limit. Thus, memory requirement is still the bottleneck that determines the feasibility limit in practice.

4.4.3 Knowledge Discovery

Finally, we applied our algorithm to a biological dataset for discovering the regulatory network responsible for controlling the expression of various genes involved in *Saccharomyces cerevisiae* (*yeast*) pheromone response pathways (Hartemink, 2001). This data set consists of 33 variables, of which 32 variables represent discretized levels of gene expression and an additional binary variable represents the mating type of various haploid strains of yeast. A total number of 320 observations are recorded. Bayesian network structure models for this data set have been constructed by using model selection methods such as greedy hill climbing, simulated annealing or by Bayesian model averaging over models selected during the simulated annealing (Hartemink, 2001; Hartemink et al., 2002).

We used our ParaREBEL algorithm to compute the exact posterior probabilities of all 1056 potential edges. The total running time was 1542 seconds on 2048 cores. We then constructed a network that consisted of (important) edges whose posteriors were greater than 0.1 (we set this threshold such that the constructed network is a DAG). The network model consists of 60 edges and is illustrated in Figure 4.12. Nodes have been augmented with color information to indicate the different groups of variables with known relationships in the literature. Edges are formatted according to their posterior probabilities.

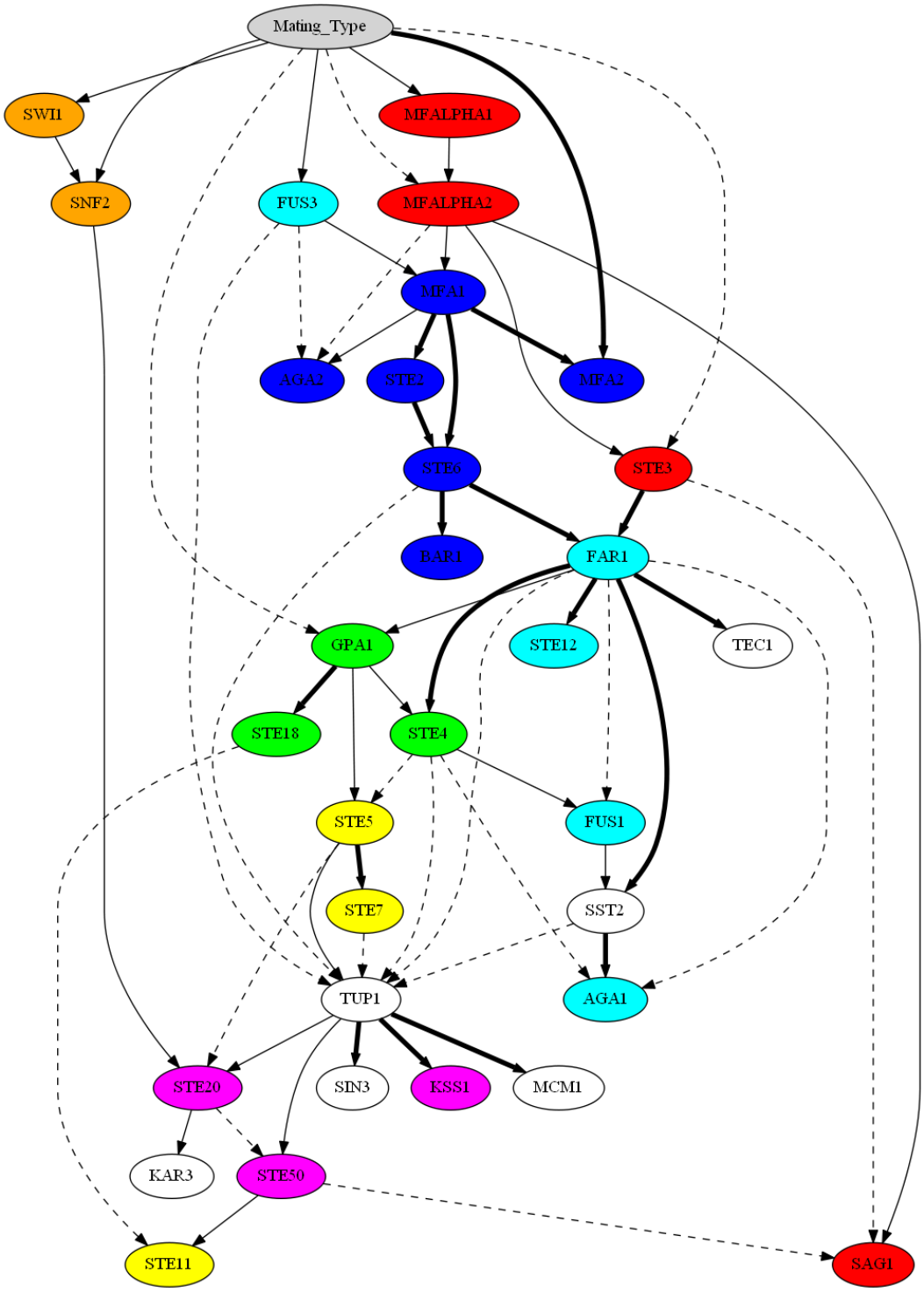


Figure 4.12: Network model learned for the *yeast* pheromone response pathways data set. Nodes have been augmented with color information to indicate the different groups of variables with known relationships in the literature. Directed edges are formatted according to their posterior probabilities: heavily weighted ($posterior \geq 0.9$), solid ($0.5 \leq posterior < 0.9$), and dashed ($0.1 \leq posterior < 0.5$).

Since the ground truth network is unknown, we cannot evaluate the accuracy of the model. However, we observe a number of interesting properties. First, variables in the same group (with the same color) tend to form a cluster (directly connected subgraph) in the network and the intra-class edges are generally more probable than the inter-class edges. This demonstrates that our algorithm is capable of recovering the (important) interactions in the *yeast* pheromone response pathways. Second, the `Mating_Type` variable is at the source of the network, and contributes to the ability to predict the state of a large number of variables, which is to be expected. Further, in (Hartemink, 2001), two types of models were learned, one obtained using greedy or simulated annealing search without any domain constraint (see Figure 7-3 in (Hartemink, 2001)), the other learned using the similar search approaches but with constraints governing the inclusion and exclusion of edges which were derived from genomic analysis (see Figure 7-4 in (Hartemink, 2001)). Interestingly, our network, which was constructed without any domain constraints, is more like the model learned with the constraints. This suggests that the network constructed with edge posteriors may achieve better modeling of the regulatory network than the model learned using model selection methods. Future research could explore additional data sets to confirm this observation.

4.5 Discussion and Conclusion

Exact Bayesian structure discovery in Bayesian networks requires exponential time and space. In this chapter, we have presented a parallel algorithm capable of computing the exact posterior probabilities for all $n(n-1)$ potential edges with optimal time and space efficiency. To our knowledge, this is the first practical parallel algorithm for computing the exact posterior probabilities of structural features in Bayesian networks. We demonstrated its capability on datasets with up to 33 variables and its scalability on up to 2048 processors. To our knowledge, 33-variable network is the largest problem solved so far. We have also applied our algorithm to a biological data set for discovering the (*yeast*) pheromone response pathways. This demonstrated our algorithm in the task of knowledge discovery.

Our algorithm makes twofold algorithmic contributions. First, it achieves an efficient parallelization of the base serial algorithm by presenting a delicate way to coordinate the computa-

tions of correlated DP procedures such that large amount of data exchange is suppressed during the transitions between these DP procedures. Second, it develops two parallel techniques for computing two variants of well-known *zeta transform*. These features or ideas can potentially be extended and applied in developing parallel algorithms for related problems. For example, the algorithm in (Tian and He, 2009) involves similar steps and transforms. Further, as zeta transforms are fundamental objects in combinatorics and algorithmics, the parallel techniques developed here would also benefit the researches beyond the context of Bayesian networks (Björklund et al., 2007, 2010; Nederlof, 2009).

From the experiments, we observed that memory requirement reached the limit much faster than computation time did. Thus, one of the future work is to improve the algorithm such that less space is used. Particularly, there is a possibility to combine the present algorithm with the method in (Parviainen and Koivisto, 2010) to trade space against time.

CHAPTER 5. EXACT BAYESIAN LEARNING OF ANCESTOR RELATIONS

In chapter 4, we presented a parallel algorithm for computing the exact posterior probabilities of all directed edges in Bayesian networks. Our parallel algorithm is based on Koivisto (2006a)'s DP algorithm, which can only evaluate the modular structural features, e.g., edges. To deal with non-modular feature, e.g., ancestor relations, an analogous DP algorithm takes $O(n3^n)$ time and $O(3^n)$ space (Parviainen and Koivisto, 2011). However, their algorithm requires a special form of structure prior over DAGs that does not respect Markov equivalence. In this chapter, we develop a new DP algorithm for exact Bayesian learning of ancestor relations. Unlike the DP algorithm by Parviainen and Koivisto (2011), our algorithm uses the standard structure-modular prior, thus allows the uniform prior and respects Markov equivalence.

5.1 Introduction

Ancestor relations, defined as a directed path in Bayesian networks, encode long-range causal relations between variables. For example, biologists would also be interested in identifying all upstream activators of a target gene in a gene regulatory network in addition to its direct regulators. As mentioned, inferring the existence of an ancestor relation based on a single DAG is unreliable. Instead, we take the Bayesian approach and try to compute the posterior probability of the ancestor relation by integrating over all possible DAGs.

Computing the posterior probability of ancestor relations is harder, because ancestor relations are non-modular features whose representations can not be factorized like the modular features. Parviainen and Koivisto (2011) proposed a DP algorithm that can compute the posteriors of all possible ancestor relations in $O(n3^n)$ time and $O(3^n)$ space. Their algorithm is

analogous to Koivisto (2006a)'s algorithm for computing edge posteriors. One issue with these algorithms is that they all assume an order-modular prior $P(G)$, thus perform summation over order space instead of DAG space. As a result, the computed posteriors would bias towards DAGs consistent with more linear orders and the Markov equivalence is not respected either. To adhere to the uniform prior, Tian and He (2009) developed a novel DP algorithm directly summing over the DAG space. Their algorithm is capable of evaluating all directed edges in $O(n3^n)$ time and $O(n2^n)$ space.

In this chapter we extend Tian and He (2009)'s work and develop a novel algorithm to compute the exact posterior probabilities of ancestor relations (directed paths) in Bayesian networks.

5.2 Preliminaries

Given an observational data D, the joint probability $P(G, D)$ is composed of

$$P(G, D) = P(G)P(D|G), \tag{5.1}$$

where $P(G)$ specifies the structure prior, and $P(D|G)$ is the data likelihood.

With standard assumptions on the parameter priors (Dirichlet prior for multinomial random variables, Wishart prior for Gaussian random variables) including global and local parameter independence and parameter modularity (Cooper and Herskovits, 1992; Friedman and Koller, 2003), the data likelihood $P(D|G)$ is decomposed into

$$P(D|G) = \prod_{i \in V} score_i(Pa_i^G : D), \tag{5.2}$$

where $score_i(Pa_i^G : D)$ is the so-called local scores and has a closed-form solution.

Moreover, the *structure modularity* assumes

$$P(G) = \prod_{i \in V} Q_i(Pa_i^G), \tag{5.3}$$

where $Q_i(Pa_i^G)$ is some function from the subsets of $V - \{i\}$ to the non-negative reals. For ease of exposition, we define, for any $i \in V$ and $Pa_i^G \subseteq V - \{i\}$

$$B_i(Pa_i^G) \equiv Q_i(Pa_i^G) score_i(Pa_i^G : D). \tag{5.4}$$

We then obtain

$$P(G, D) = \prod_{i \in V} B_i(Pa_i^G). \tag{5.5}$$

5.3 Previous Approaches

Let f be a structural feature represented by an indicator function such that $f(G)$ is 1 if the feature is present in G and 0 otherwise. In Bayesian approach, we are interested in computing the posterior $P(f|D)$ of the feature, which can be obtained by computing the joint probability $P(f, D)$ as

$$P(f, D) = \sum_G f(G)P(G, D). \tag{5.6}$$

The summation is intractable in practice since the number of all possible DAGs is in the order of $O(n!2^{n(n-1)/2})$. Thus, much research has proposed to work on the order space (Friedman and Koller, 2003; Koivisto and Sood, 2004; Koivisto, 2006a; Parviainen and Koivisto, 2011). Formally, an order $\prec$ is a linear order $(L_1, ..., L_n)$ on the index set V, where L_i specifies the predecessors of i in the order, i.e., $L_i = \{j : j \prec i\}$. We say that a structure $G = (Pa_1, ..., Pa_n)$ is consistent with an order $\prec$, denoted by $G \in \prec$, if $Pa_i \subseteq L_i$ for all i. Then we can compute

$$P(f, D) = \sum_{\prec} P(\prec) \sum_{G \in \prec} f(G)P(D|G)P(G| \prec). \tag{5.7}$$

It turns out with such treatment, the computation can be more efficient and convenient. Indeed, it has been shown that the posteriors of all possible ancestor relations can be evaluated in time $O(n3^n)$ and space $O(3^n)$ using this order-based summation scheme (Parviainen and Koivisto, 2011).

This treatment is problematic because it treats different variable orders as mutually exclusive events. However, the corresponding sets of consistent DAGs are overlapping. If we introduce a uniform prior $P(\prec)$ and a uniform $P(G| \prec)$, the resulting prior $P(G)$ is not uniform. It weights the DAGs by the number of linear extensions. For example, an empty network without any edge is consistent with $n!$ linear orders, while a chain network (see Figure 5.1a) is consistent with only one linear order. The resulting posteriors will bias towards DAGs consistent with more linear orders. For the same reason, two Markov equivalent DAGs (Pearl,

2000) may receive unequal priors. For example, the two DAGs shown in Figure 5.1 are Markov equivalent. However, the tree DAG in Figure 5.1b will be weighted 6 times as the chain DAG in Figure 5.1a. Thus, the Markov equivalence is not respected.

(a) A chain DAG (b) A tree DAG

Figure 5.1: Two Markov equivalent DAGs.

Next, we will develop a novel algorithm for Bayesian learning of ancestor relations that directly performs summation over the DAG space by exploiting sinks. Our algorithm allows the uniform prior $P(G)$ and respects the Markov equivalence.

5.4 Bayesian Learning of Ancestor Relations

5.4.1 Algorithm

We say s is an ancestor of t, or t is a descendant of s, if G contains a directed path from s to t, denoted as $s \rightsquigarrow t$. The posterior probability of an ancestor relation $s \rightsquigarrow t$ is evaluated by

$$P(s \rightsquigarrow t|D) = P(s \rightsquigarrow t, D)/P(D). \tag{5.8}$$

The joint probability $P(s \rightsquigarrow t, D)$ can be computed by

$$P(s \rightsquigarrow t, D) = \sum_{G \in \mathcal{G}_{s \rightsquigarrow t}} P(G, D) = \sum_{G \in \mathcal{G}_{s \rightsquigarrow t}} \prod_{i \in V} B_i(Pa_i^G), \tag{5.9}$$

where $\mathcal{G}_{s \rightsquigarrow t} \equiv \{G : s \rightsquigarrow t \in G\}$, namely the set of all possible DAGs over V that contain a $s \rightsquigarrow t$.

For any $S \subseteq V$, let G_S denote a DAG over S. For any $v \in S$, let $Pa_v^{G_S}$ be the parent set of v in G_S, and $de_{G_S}(v) \equiv \{u|u \leftarrow \cdots \leftarrow v \text{ in } G_S \text{ or } u = v\}$ be the set of all descendants of v (including v) in G_S. For any T, S such that $s \in T \subseteq S \subseteq V$, let $\mathcal{G}_s(S, T)$ denote the set of all possible DAGs over S such that T are the set of all descendants of s in G_S. That is, $G_S \in \mathcal{G}_s(S, T)$ if and only if $de_{G_S}(s) = T$. We define, for any $s \in T \subseteq S \subseteq V$,

$$H_s(S, T) \equiv \sum_{G_S \in \mathcal{G}_s(S,T)} \prod_{i \in S} B_i(Pa_i^{G_S}). \tag{5.10}$$

Then we have the following lemma:

Lemma 5.1.

$$P(s \rightsquigarrow t, D) = \sum_{T:\{s,t\}\subseteq T\subseteq V} H_s(V,T). \tag{5.11}$$

Proof. We have $\mathcal{G}_{s\rightsquigarrow t} = \cup_{T:\{s,t\}\subseteq T\subseteq V}\mathcal{G}_s(V,T)$. Further, for any $T_1 \neq T_2$, we have $\mathcal{G}_s(V,T_1) \cap \mathcal{G}_s(V,T_2) = \emptyset$. This means $\mathcal{G}_s(V,T)$ for all T such that $s,t \in T \subseteq V$ form a partition of the set $\mathcal{G}_{s\rightsquigarrow t}$. An illustration of this partition is showed in Figure 5.2. Thus,

$$P(s \rightsquigarrow t, D) = \sum_{G\in\mathcal{G}_{s\rightsquigarrow t}} \prod_{i\in V} B_i(Pa_i^G) = \sum_{T:\{s,t\}\subseteq T\subseteq V} \sum_{G\in\mathcal{G}_s(V,T)} \prod_{i\in V} B_i(Pa_i^G) = \sum_{T:\{s,t\}\subseteq T\subseteq V} H_s(V,T). \tag{5.12}$$

□

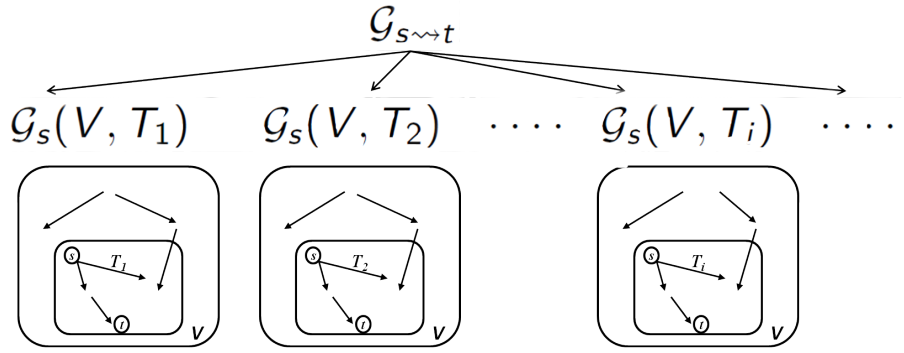


Figure 5.2: A partition of $G_{s\rightsquigarrow t}$ by s's descendant set T.

Now the problem is decomposed into computing $H_s(V,T)$ for all T s.t. $\{s,t\} \subseteq T \subseteq V$. We show that $H_s(S,T)$ for all T, S such that $\{s\} \subseteq T \subseteq S \subseteq V$ can be computed recursively. We immediately noticed that these $H_s(S,T)$'s can be divided into two cases: $T = \{s\}$ and $T \neq \{s\}$ (or $T - \{s\} \neq \emptyset$).

Case 1: $T = \{s\}$.

In this case, s is sink in G_S (see Figure 5.3). For any $S \subseteq V$, let $\mathcal{G}(S)$ denote the set of all possible DAGs over S. Then we have

$$H_s(S,\{s\}) = [\sum_{Pa_s\subseteq S-\{s\}} B_s(Pa_s)][\sum_{G_{S-\{s\}}\in\mathcal{G}(S-\{s\})} \prod_{i\in S-\{s\}} B_i(Pa_i^{G_{S-\{s\}}})]. \tag{5.13}$$

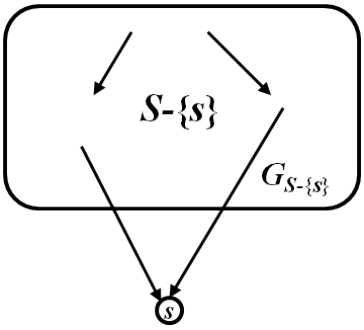


Figure 5.3: Case 1: $T = \{s\}$.

For any $S \subseteq V$, define function

$$H(S) \equiv \sum_{G_S \in \mathcal{G}(S)} \prod_{i \in S} B_i(Pa_i), \tag{5.14}$$

and for each $i \in V$ and all $U \subseteq V - \{i\}$, define

$$A_i(U) \equiv \sum_{Pa_i \subseteq U} B_i(Pa_i). \tag{5.15}$$

The function A_i is known as the zeta transform of B_i which can be computed by the so-called fast zeta transform algorithm in time $O(n2^n)$ (Koivisto and Sood, 2004). Now we can rewrite Equation 5.13 as

$$H_s(S, \{s\}) = A_s(S - \{s\})H(S - \{s\}). \tag{5.16}$$

Tian and He (2009) proposed a DP algorithm to sum over $\mathcal{G}(S)$ by exploiting possible sinks of DAGs and inclusion-exclusion principle. Due to Proposition 2 in (Tian and He, 2009), we have that $H(S)$ can be computed recursively by the following

$$H(S) = \sum_{k=1}^{|S|} (-1)^{k+1} \sum_{W \subseteq S, |W|=k} H(S - W) \prod_{j \in W} A_j(S - W), \tag{5.17}$$

with the base case $H(\emptyset) = 1$. $H(S)$ for all $S \subseteq V$ can be computed with time $O(n3^{n-1})$ and space $O(n2^n)$ (Tian and He, 2009).

Case 2: $T \neq \{s\}$ **(or** $T - \{s\} \neq \emptyset$**).**

For any $W \subseteq S$, let $\mathcal{G}_s(S, T, W)$ denote the set of DAGs in $\mathcal{G}_s(S, T)$ such that all nodes in W are (must be) sinks.[1] We first note that for any W such that $s \in W$, $\mathcal{G}_s(S, T, W) = \emptyset$. This

[1] W may not include all the sinks in G_S. Some nodes in $S - W$ could be sinks.

trivially holds because $T - \{s\} \neq \emptyset$ implies that s must have other descendants besides itself thus s cannot be a sink in G_S. Note that $\mathcal{G}_s(S, T, \emptyset) = \mathcal{G}_s(S, T)$. For any $W \subseteq S - \{s\}$, we define

$$F_s(S, T, W) \equiv \sum_{G_S \in \mathcal{G}_s(S,T,W)} \prod_{i \in S} B_i(Pa_i^{G_S}). \tag{5.18}$$

Since every DAG has at least one sink, we have $\mathcal{G}_s(S, T) = \cup_{j \in S-\{s\}} \mathcal{G}_s(S, T, \{j\})$. Further, it is clear that $\cap_{j \in W} \mathcal{G}_s(S, T, \{j\}) = \mathcal{G}_s(S, T, W)$. Then the summation over $\mathcal{G}_s(S, T)$ in Equation 5.10 can be computed by summing over the DAGs in $\mathcal{G}_s(S, T, \{j\})$ separately and correcting the overlaps. By weighted inclusion-exclusion principle,

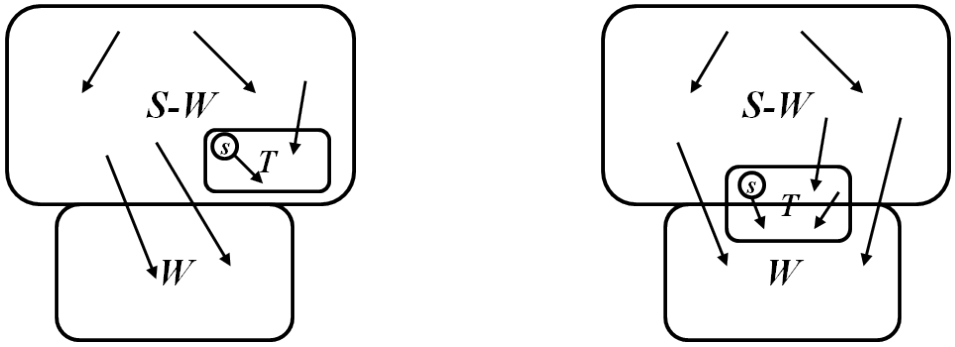
$$\begin{aligned} H_s(S, T) &= \sum_{k=1}^{|S|-1} (-1)^{k+1} \sum_{W \subseteq S-\{s\}, |W|=k} \sum_{G_S \in \mathcal{G}_s(S,T,W)} \prod_{i \in S} B_i(Pa_i^{G_S}) \\ &= \sum_{k=1}^{|S|-1} (-1)^{k+1} \sum_{W \subseteq S-\{s\}, |W|=k} F_s(S, T, W). \end{aligned} \tag{5.19}$$

Next we show that $F_s(S, T, W)$ and $H_s(S, T)$ can be computed recursively. The central idea is to convert the sum of products in Equation 5.18 to product of sums. That is, we will consider the summation over W and the summation over $S - W$ separately. Since any node in W must be a sink in G_S, it can only select parents from $S - W$. There are two sub-cases.

Sub-case 1: $T \cap W = \emptyset$.

If $T \cap W = \emptyset$, the sum of products in Equation 5.18 can be freely decomposed to product of sums for nodes in W and sum over remaining nodes in $S - W$. As showed in Figure 5.4a, any node in W can only select parents from $S - W - T$. For nodes in $S - W$, we have summation over $\mathcal{G}_s(S - W, T)$. Then we have

$$\begin{aligned} F_s(S, T, W) &= [\prod_{j \in W} \sum_{Pa_j \subseteq (S-T-W)} B_j(Pa_j)][\sum_{G_{S-W} \in \mathcal{G}_s(S-W,T)} \prod_{i \in S-W} B_i(Pa_i^{G_{S-W}})] \\ &= \prod_{j \in W} A_j(S - T - W) H_s(S - W, T) \\ &= \prod_{j \in W} A_j(S - T - W) H_s(S - W, T - W) \text{ (because } T - W = T \text{ in this case).} \end{aligned} \tag{5.20}$$



(a) Sub-case 1: $T \cap W = \emptyset$ (b) Sub-case 2: $T \cap W \neq \emptyset$

Figure 5.4: Two sub-cases when computing $F_s(S,T,W)$.

Sub-case 2: $T \cap W \neq \emptyset$.

In this case, nodes in $W-T$, $T \cap W$, and $S-W$ should be handled separately (see Figure 5.4b). Nodes in $W-T$ can only select parents from $S-W-T$. Any node in $T \cap W$ can select parents from $S-W$. In addition, at least one node from $T-W$ must be included in its parent set to guarantee that it is a descendant of s. For nodes in $S-W$, we have summation over $\mathcal{G}_s(S-W, T-W)$. Then we have

$$
\begin{aligned}
F_s(S,T,W) &= [\prod_{j\in T\cap W} \sum_{\substack{Pa_j\subseteq(S-W)\\ Pa_j\cap(T-W)\neq\emptyset}} B_j(Pa_j)][\prod_{j\in W-T} \sum_{\substack{Pa_j\subseteq\\ (S-T-W)}} B_j(Pa_j)][\sum_{\substack{G_{S-W}\in\\ \mathcal{G}_s(S-W,T-W)}} \prod_{i\in S-W} B_i(Pa_i)] \\
&= \{\prod_{j\in T\cap W} [\sum_{Pa_j\subseteq(S-W)} B_j(Pa_j) - \sum_{\substack{Pa_j\subseteq\\ (S-W-T)}} B_j(Pa_j)]\} \prod_{j\in W-T} A_j(S-T-W) H_s(S-W,T-W) \\
&= \{\prod_{j\in T\cap W} [A_j(S-W) - A_j(S-W-T)]\} \prod_{j\in W-T} A_j(S-T-W) H_s(S-W,T-W).
\end{aligned}
\tag{5.21}
$$

For ease of exposition, define function $\mathcal{A}_s$ as follows:

$$
\mathcal{A}_s(S,T,W) \equiv \begin{cases} \prod_{j\in W} A_j(S-T-W) & \text{if } T\cap W = \emptyset \\ \{\prod_{j\in T\cap W} [A_j(S-W) - A_j(S-W-T)]\} \prod_{j\in W-T} A_j(S-T-W) & \text{if } T\cap W \neq \emptyset \end{cases}
\tag{5.22}
$$

Now combining Sub-case 1 and 2, $F_s(S,T,W)$ can be neatly written as

$$
F_s(S,T,W) = \mathcal{A}_s(S,T,W) H_s(S-W,T-W) \tag{5.23}
$$

Plugging Equation 5.23 into Equation 5.19, we obtain

$$H_s(S,T) = \sum_{k=1}^{|S|-1} (-1)^{k+1} \sum_{\substack{W \subseteq S-\{s\} \\ |W|=k}} \mathcal{A}_s(S,T,W) H_s(S-W, T-W) \tag{5.24}$$

In summary, we arrive at the following recursive scheme for computing $H_s(S,T)$ for any $\{s\} \subseteq T \subseteq S \subseteq V$.

Theorem 5.1. *For all T, S such that $\{s\} \subseteq T \subseteq S \subseteq V$, $H_s(S,T)$ can be computed recursively as follows:*

(1) For all $S \subseteq V - \{s\}$,

$$H(S) = \sum_{k=1}^{|S|} (-1)^{k+1} \sum_{W \subseteq S, |W|=k} H(S-W) \prod_{j \in W} A_j(S-W),$$

with the base case $H(\emptyset) = 1$.

(2) For all S s.t. $s \in S \subseteq V$, $H_s(S, \{s\}) = A_s(S - \{s\}) H(S - \{s\})$.

(3) For all T, S s.t. $\{s\} \subset T \subseteq S \subseteq V$,

$$H_s(S,T) = \sum_{k=1}^{|S|-1} (-1)^{k+1} \sum_{\substack{W \subseteq S-\{s\} \\ |W|=k}} \mathcal{A}_s(S,T,W) H_s(S-W, T-W).$$

5.4.2 Efficient Computation of $\mathcal{A}_s(S,T,W)$

We note that there are repeated computation of $\prod_{j \in W} A_j(U)$ in the phase of computing $H(S)$ and in the phase of computing $H_s(S,T)$. To facilitate the computation of $\prod_{j \in W} A_j(U)$ and $\mathcal{A}_s(S,T,W)$, we define for any $W \subseteq V$, $U \subseteq V - W$,

$$AA(U,W) \equiv \prod_{j \in W} A_j(U). \tag{5.25}$$

Then for a fixed U, we have

$$AA(U,W) = A_j(U) AA(U, W - \{j\}) \text{ for any } j \in W. \tag{5.26}$$

Thus, for a fixed U, $AA(U,W)$ for all $W \subseteq V - U$ can be computed in the manner of dynamic programming in $O(2^{n-|U|})$ time. Then $AA(U,W)$ for all $U \subseteq V$ and all $W \subseteq V - U$ can

be computed in $\sum_{|U|=0}^{n} \binom{n}{|U|} 2^{n-|U|} = O(3^n)$ time. With the pre-computation of $AA(U, W)$, $\mathcal{A}_s(S, T, W)$ for any T, S such that $\{s\} \subset T \subseteq S \subseteq V$ can be computed more efficiently: if $T \cap W = \emptyset$,

$$\mathcal{A}_s(S, T, W) = \prod_{j \in W} A_j(S - T - W) = AA(S - T - W, W), \tag{5.27}$$

else if $T \cap W \neq \emptyset$,

$$\begin{aligned} \mathcal{A}_s(S, T, W) &= \{ \prod_{j \in T \cap W} [A_j(S - W) - A_j(S - W - T)] \} \prod_{j \in W - T} A_j(S - T - W) \\ &= \{ \prod_{j \in T \cap W} [AA(S - W, \{j\}) - AA(S - W - T, \{j\})] \} AA(S - T - W, W - T). \end{aligned} \tag{5.28}$$

In summary, we have

$$\mathcal{A}_s(S, T, W) \equiv \begin{cases} AA(S - T - W, W) & \text{if } T \cap W = \emptyset \\ \{\prod_{j \in T \cap W} [AA(S - W, \{j\}) - AA(S - W - T, \{j\})]\} & \\ AA(S - T - W, W - T) & \text{if } T \cap W \neq \emptyset \end{cases} \tag{5.29}$$

5.4.3 Overall Algorithm to Compute $P(s \rightsquigarrow t|D)$

Finally we summarize the results in section 5.4.1 and section 5.4.2 and outline the algorithm for computing posterior probability of any ancestor relation $s \rightsquigarrow t$.

Algorithm 5.1 Computing the posterior probability of an ancestor relation $s \rightsquigarrow t$.

(a) For all $i \in V$, $Pa_i \subseteq V - \{i\}$, compute $B_i(Pa_i)$. Time complexity $O(n2^{n-1})$.

(b) For all $i \in V$, $U \subseteq V - \{i\}$, compute $A_i(U)$. Time complexity $O(n2^{n-1})$.

(c) For all $U \subseteq V$, $W \subseteq V - U$, compute $AA(U, W)$. Time complexity $O(3^n)$.

(d) For all $S \subseteq V$, compute $H(S)$ in the lexicographic order of S. Time complexity $O(3^{n-1})$.

(e) For all $S \subseteq V$ s.t. $s \in S$, compute $H_s(S, \{s\})$. Time complexity $O(2^{n-1})$.

(f) For all T, S such that $\{s\} \subset T \subseteq S \subseteq V$, compute $H_s(S,T)$ in the lexicographic order of S and T, with T as the outer loop and S as the inner loop. For example, we start the computation of $H_s(S,\{s,i\})$ for each $i \in V - \{s\}$ and all S such that $\{s,i\} \subseteq S \subseteq V$ in the lexicographic order of S. Then we compute $H_s(S,\{s,i,j\})$ for each $\{i,j\} \subseteq V - \{s\}$ and all S such that $\{s,i,j\} \subseteq S \subseteq V$ in the lexicographic order of S, so on and so forth and finally we compute $H_s(V,V)$.

(g) Compute $P(s \rightsquigarrow t, D)$ by $P(s \rightsquigarrow t, D) = \sum_{T:\{s,t\}\subseteq T\subseteq V} H_s(V,T)$ and output $P(s \rightsquigarrow t|D) = P(s \rightsquigarrow t, D)/H(V)$.[2] Time complexity $O(2^{n-2})$.

It is worth mentioning that the posterior probability of any ancestor relation can only be interpreted with regard to its prior probability. This prior probability can also be computed by Algorithm 5.1 with all local scores $score_i(Pa_i^G : D)$ set to 1.

5.4.4 Time and Space Complexity

The computing times for steps (a) to (e) and step (g) have been given already. The overall computing time is actually dominated by step (f).

For any T, S s.t. $\{s\} \subset T \subseteq S \subseteq V$, we compute $H_s(S,T)$ in $O(|S| \cdot 2^{|S|-1})$ time (any $\mathcal{A}_s(S,T,W)$ can be computed on the fly in time $O(|S|)$). Thus, all $H_s(S,T)$ can be computed in time

$$\begin{aligned}
&\sum_{|S|=2}^{n} \left\{ \binom{n-1}{|S|-1} \left[\sum_{|T|=2}^{|S|} \binom{|S|-1}{|T|-1} |S| \cdot 2^{|S|-1} \right] \right\} \\
&= \sum_{|S|=2}^{n} \left\{ \binom{n-1}{|S|-1} \left[|S| \cdot 2^{|S|-1} \sum_{|T|=2}^{|S|} \binom{|S|-1}{|T|-1} \right] \right\} \\
&= \sum_{|S|=2}^{n} \left[\binom{n-1}{|S|-1} |S| \cdot 2^{|S|-1} \cdot 2^{|S|-1} \right] \\
&= \sum_{|S|=2}^{n} \left[\binom{n-1}{|S|-1} |S| \cdot 4^{|S|-1} \right] = n5^{n-1}.
\end{aligned}$$

Thus, the total computation time is $O(n5^{n-1} + 3^n + n2^{n-1}) = O(n5^{n-1})$.

[2] It has been shown by Tian and He (2009) that $P(D) = H(V)$.

To compute the posterior probabilities for all node pairs s, t, it suffices to repeat the computation step (e) and step (f) for each $s \in V$, and for a given s to repeat step (g) for each t. Thus, the total time for computing all possible ancestor relations $s \rightsquigarrow t$ is $O(n^2 5^{n-1})$.

$B_i(Pa_i)$ for all $i \in V$, $Pa_i \subseteq V - \{i\}$ take $O(n2^{n-1})$ space. $A_j(U)$ for all $j \in V$, $U \subseteq V - \{j\}$ take $O(n2^{n-1})$ space. $AA(U, W)$ for all $U \subseteq V$, $W \subseteq V - U$ consume $\sum_{|U|=0}^{n} \binom{n}{|U|} 2^{n-|U|} = O(3^n)$ space. $H(S)$ for all $S \subseteq V$ take $O(2^n)$ space. $H_s(S, T)$ for all $\{s\} \subseteq T \subseteq S \subseteq V$ consume $\sum_{|S|=1}^{n} \binom{n-1}{|S|-1} 2^{|S|-1} = O(3^{n-1})$ space. Since each step in Algorithm 5.1 relies only on the previous step. We need only store relevant scores in the memory. That is, after we compute all $A_i(U)$ scores, we can immediately delete all $B_i(Pa_i)$ scores; after computing all $AA(U, W)$ scores, we delete $A_i(U)$ scores. When computing step (f), we need only keep $AA(U, W)$ and $H_s(S, T)$ scores in memory. In such way, we can use the memory more efficiently. The space requirement is therefore $O(3^n + n2^n)$.

In summary, we have the following theorem.

Theorem 5.2. *The posterior probability for any ancestor relation $s \rightsquigarrow t$ can be computed in $O(n5^{n-1})$ time and $O(3^n + n2^n)$ space. The posterior probabilities for all $n(n-1)$ possible ancestor relations can be computed in $O(n^2 5^{n-1})$ time and $O(3^n + n2^n)$ space.*

5.4.5 Exact Bayesian Learning of $s \rightsquigarrow p \rightsquigarrow t$ Relations

It turns out that the techniques for computing $s \rightsquigarrow t$ relations can be extended to compute the posteriors of $s \rightsquigarrow p \rightsquigarrow t$ relations, i.e., a directed path from s to t via p. For example, biologists are interested in whether the influence of a gene on a downstream gene is regulated by some intermediate gene or factor. Learning this type of structural features is therefore of great interests. Due to the space limit, we only present our conclusion here in Theorem 5.3. The algorithm and proofs are included in Appendix B.

Theorem 5.3. *The posterior probability of any $s \rightsquigarrow p \rightsquigarrow t$ relation can be computed in $O(n7^{n-2})$ time and $O(4^{n-2} + 3^n)$ space.*

5.5 Experiments

We have implemented Algorithm 5.1 in C++. We used BDe score for $score_i(Pa_i : D)$ with equivalent sample size being 1 (Heckerman and Chickering, 1995). We applied uniform structure prior $P(G)$ by setting all $Q_i(Pa_i)$'s to be 1.[3] All experiments were done on a Linux desktop PC with 3.33 GHz Intel Core2 Duo CPU and 4 GB memory.

5.5.1 Running Times

We first examine the running times of our algorithm on several data sets from the UCI Machine Learning Repository. The results are presented in Table 5.1, where n is the number of variables, m is the sample size of each data set, $T(B)$ records the time for computing all $B_i(Pa_i)$'s, i.e., the local scores, and $T(total)$ is the total time for evaluating all $n(n-1)$ ancestor relations. We clearly see that the running times are reflective of the exponential dependence on n with a base around 5. This is consistent with Theorem 5.2.

Table 5.1: Execution time (in seconds)

Data Sets	n	m	$T(B)$	$T(total)$
Weather	5	14	3e-4	7e-4
Iris	5	150	6e-4	6e-4
Asia	8	500	0.02	0.2
Tic-Tac-Toe	10	958	0.6	6.5
CYTO	11	5400	8.4	32
Wine-11	11	178	0.8	76
Wine-12	12	178	1.8	411
Wine-13	13	178	4.6	2331
Wine	14	178	11.6	12856

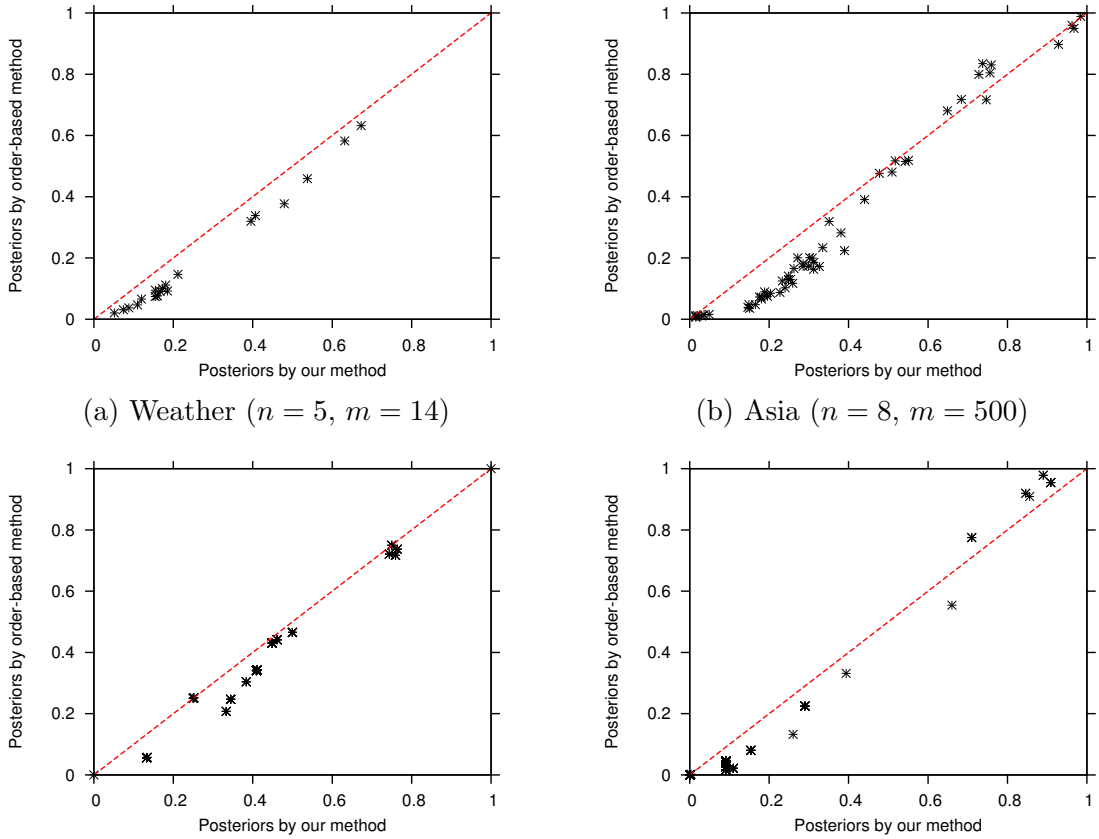
5.5.2 Comparison of Posteriors

The order-based approach by Parviainen and Koivisto (2011) and our approach differ in the structure prior $P(G)$ assigned to DAGs. The order-based approach places a non-uniform prior

[3] Note that a constant $P(G)$ could be canceled out in computing $P(f|D) = P(f, D)/P(D)$.

over DAGs, favoring DAGs that are consistent with more linear orders, while our approach adheres to the uniform prior. Here we compare the posteriors computed by the two approaches on four different data sets in Figure 5.5.

We can see that the posteriors computed by the two approaches differ in most of the cases, demonstrating the non-negligible effect of priors on the computation results. Further, we observed that the order-based approach often underestimates the posteriors (see Figure 5.5a and Figure 5.5c). This can be understood by noticing that DAGs consistent with more linear orders usually have simpler structures, for example, fewer edges or fewer directed paths than those DAGs consistent with fewer linear orders. Since DAGs consistent with fewer linear orders receive less weights in the order-based approach, the ancestor relations implied by these DAGs are undercounted.



(a) Weather ($n = 5$, $m = 14$)

(b) Asia ($n = 8$, $m = 500$)

(c) Tic ($n = 10$, $m = 958$)

(d) Wine ($n = 12$, $m = 178$)

Figure 5.5: Scatter plots that compare posteriors of ancestor relations computed by our algorithm and by order-based algorithm.

5.5.3 Knowledge Discovery

Finally, we applied our algorithm to a biological data set (CYTO) which consists of flow cytometry measurements of $n = 11$ phosphorylated proteins and phospholipids under 7 different interventions and 2 unperturbed conditions. 600 measurements are taken in each condition yielding a total dataset of $m = 5400$ samples. The data have been discretized into 3 states, representing low, medium and high activity according to (Sachs et al., 2005). Figure 5.6 shows a currently accepted consensus network.

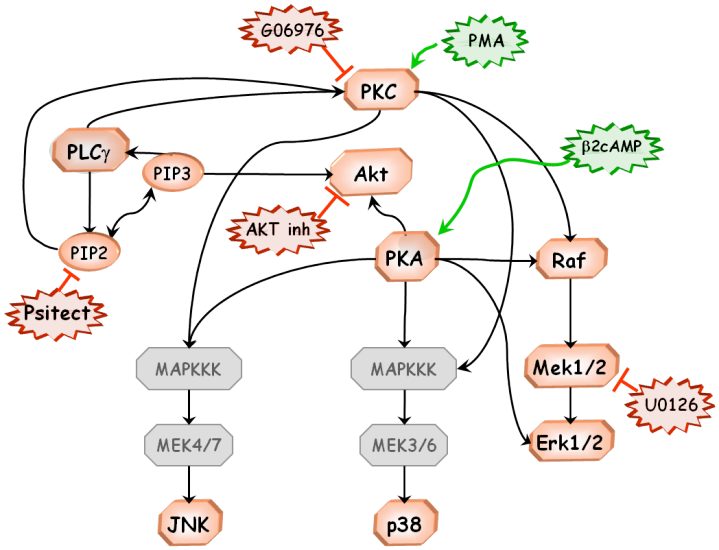


Figure 5.6: Classical model of the CYTO data set. Modified from (Sachs et al, 2005). The proteins of interest are in highlighted red rectangles, i.e., PKC, PLCγ, PIP2, PIP3, Akt, PKA, JNK, p38, Raf, Mek, Erk. Ovals with serrated edges represent various interventions (green=activators, red=inhibitors).

Table 5.2: Ancestor relations learned for CYTO data set

Source	Sinks
Raf	**Mek**, PLCγ, PIP2, PIP3, **Erk**, Akt, PKA, p38, JNK
Mek	PLCγ, PIP2, PIP3, **Erk**, Akt, PKA, p38, JNK
PLCγ	**PIP2**, **PIP3**
PIP2	**PIP3**
Erk	Akt
PKA	PLCγ, PIP2, PIP3, **Erk**, **Akt**, **p38**, **JNK**
PKC	**Raf**, **Mek**, PLCγ, PIP2, PIP3, **Erk**, Akt, PKA, **p38**, **JNK**
JNK	PLCγ, PIP2, PIP3, p38

We then used our algorithm to compute the posteriors of all 110 possible ancestor relations. We modified the local likelihood scores $B_i(Pa_i)$ to take into account the interventional nature of the data as in (Tian and Pearl, 2001). Our results shows that among all 110 possible ancestor relations, 42 ancestor relations have posteriors greater than 0.95, while all other ancestor relations have posteriors less than 0.03.[4] Table 5.2 tabulates the 42 most probable ancestor relations. Proteins are made bold if the corresponding ancestor relations also exist in the classical model. The learning results exhibit a high false positive rate if the classical model is really the true model. This may be because that the ancestor relation learning is very sensitive to the local errors. For example, one flipped edge can lead to a large number of ancestor relation errors. However, the classical model is not necessarily the true model. As our results showed, we had high certainty on the presence of each important ancestor relation ($\hat{P}(s \rightsquigarrow t|D) \geq 0.95$). Thus, it is possible that the ancestor relations discovered by our method but not in the classical model suggest potential protein signaling pathways that have yet to be discovered by biologists.

We also compared our direct learning of ancestor relations to the deduction of (important) ancestor relations from the edge posteriors. We used the algorithm by Tian and He (2009), which requires $O(n3^n)$ time and $O(n2^n)$ space, to compute the posteriors of all 110 possible *edges*. We then constructed a network that consisted of edges whose posteriors were greater than 0.5 and inferred the ancestor relations from this network. We observed that the set of (important) ancestor relations deduced from the most likely edges were exactly the same as those predicted by the direct learning. This suggests that the two approaches do not differ significantly in predicting the most significant ancestor relations, at least on this CYTO data set. More systematic evaluation will be needed to confirm this observation. Moreover, we observed that computing all edge posteriors took about 9 seconds, much faster than computing the posteriors of ancestor relations (32 seconds, see Table 5.1). However, direct learning of ancestor relation outputs the ancestor posterior probabilities while the network constructed from most likely edges does not provide such information.

[4]Note that the prior of an ancestor relation is 0.45 for $n = 11$ with the uniform structure prior.

5.6 Conclusion

In this chapter, we have developed a new DP algorithm to compute the exact posteriors of all possible ancestor relations in Bayesian networks. Compared to previous order-based algorithm, our algorithm respects the uniform structure prior and the Markov equivalence. Experimental comparison showed the order-based approach tends to underestimate the posteriors. We have also applied our algorithm to a biological data set to discover protein signaling pathways. This demonstrated our algorithm in the task of knowledge discovery. Further, we have developed an algorithm to compute the exact posterior of any $s \rightsquigarrow p \rightsquigarrow t$ relation, i.e., a directed path from s to t via p.

One major limitation of the exact algorithms proposed here (and in previous work) is their exponential complexities, which prevent their practical use for large networks. To circumvent the limitation, approximate methods such as MCMC sampling are commonly used. One potential application of the exact algorithms given in this chapter is to assess the approximate quality of approaches such as MCMC sampling.

CHAPTER 6. JOINT DISCOVERY OF SKILL PREREQUISITE GRAPHS AND STUDENT MODELS

In previous chapters, we studied how to learn Bayesian network structures from data in more accurate and efficient ways. We also showed some preliminary applications of our approaches in systems biology for modeling gene regulatory networks and protein signaling pathways. In this chapter, we show an application of Bayesian networks in the field of educational data mining. We use a Bayesian network structure to model the prerequisite relationships between the skills and study how we learn these relationships from student performance data. Further, since this Bayesian network also represent a joint probability distribution over the skill variables (and item variables), it becomes a type of student model that can be used for cognitive diagnosis. In summary, we introduce a novel algorithm that can jointly discover a prerequisite graphs and a student model from data.

6.1 Introduction

Course *curricula* are usually organized in a meaningful sequence that evolves from relatively simple lessons to more complex ones. Among these lessons, some are required to be mastered by students before the subsequent ones can be learned. For instance, students have to know how to do addition before they learn to do multiplication. We refer to *prerequisite structure* as the relationships among skills that place strict constraints on the order in which skills can be acquired.

Prerequisite structures are crucial for designing intelligent tutoring systems that assess student knowledge or offer remediation interventions to students. Building such systems require prerequisite information that is often hand-engineered by subject-matter experts in a costly

and time-consuming process. Moreover, the prerequisite structures specified by the experts are seldom tested and might be unreliable in the sense that experts may have "blind spots".

Recent interest in computer assisted education promises large amounts of data from students solving *items*— questions, problems, parts of questions. Performance data –what items a learner answers correctly– can be used to create *student models*. These models represent an estimate of skill proficiency at a given point in time (VanLehn, 1988). For example, a student model can represent that Alice has already mastered integer addition, but Bob has not. Student models are often used to personalize instruction in tutoring systems or to predict future student performance. In this chapter, we introduce *Combined student Modeling and prerequisite Discovery* (COMMAND), a novel algorithm for simultaneously discovering prerequisite structure of skills and a student model from student performance data.

6.2 Relation to Prior Work

Prior work has investigated how to discover prerequisites among items without considering their mapping into skills (Desmarais et al., 2006; Vuong et al., 2010). Item-to-skill mappings (also called Q-matrices) are desirable because they allow more interpretable diagnostic information. Because of this, follow-up work (Brunskill, 2010; Chen et al., 2015) has studied whether a pair of skills have a prerequisite relationship or not. For this, we can measure if a model that assumes a dependency between the two skills explains the data better than a model that assumes independence. This comparison can be done with data likelihood (Brunskill, 2010) or association rule mining (Chen et al., 2015). Although promising, prior methods have limitations that we address:

1. We estimate the global prerequisite structure, not just the pairwise relationships. For example, suppose we want to discover the prerequisites of three skills for English learning (S_1:syntax, S_2:cohesion and S_3:lexical rules). If we use prior methods, we discover that the three skills are related among each other. However, pairwise methods are unable to tell if the relationships are due to indirect (e.g, $S_3 \rightarrow S_2 \rightarrow S_1$), or direct (e.g, $S_3 \rightarrow S_2 \rightarrow S_1$) effects.

2. It is unclear how to use the output of these prerequisite structures for student modeling. For example, it is not obvious how to best use them to make predictions of future student performance.

3. Prior work does not provide quantitative evaluation using real student data. Overall, learner data has been used to provide examples, but without any methodology that can help compare what algorithm works better.

Bayesian networks have been useful to model prerequisite structures (Mislevy et al., 1999). Bayesian networks allows modeling the full structure of skills (beyond pairwise relationships) and can encode conditional independence between the skills. Unfortunately, prior work with Bayesian networks requires a domain expert to design the prerequisite structures (Käser et al., 2014), and automatic techniques have not been demonstrated with real student data (Scheines et al., 2014). We now describe the COMMAND algorithm that discovers a Bayesian network that encodes the prerequisite structure of skills.

6.3 The COMMAND Algorithm

COMMAND learns the prerequisite structure of the skills from data with a statistical model called Bayesian network (Pearl, 1988; Spirtes et al., 2001). Bayesian networks are one type of probabilistic graphical models because they can be represented visually and algebraically as a collection of nodes and edges. A tutorial description of Bayesian networks in education can be found elsewhere (Mislevy et al., 1999), but for now we say that they are often described with two components: the nodes represent the random variables, which we describe using *conditional probability tables* (CPTs), and the set of edges that form a *directed acyclic graph* (DAG) represent the conditional dependencies between the variables. Bayesian networks are a flexible tool that can be used to model an entire curriculum.

Figure 6.1 illustrates an example of a prerequisite structure modeled with a Bayesian network. Here, we relate four test items with the skills of addition and multiplication. Addition is a prerequisite of multiplication thus there is an arrow from addition to multiplication. Modeling prerequisites as edges in a Bayesian network allows us to frame the discovery of the prerequisite

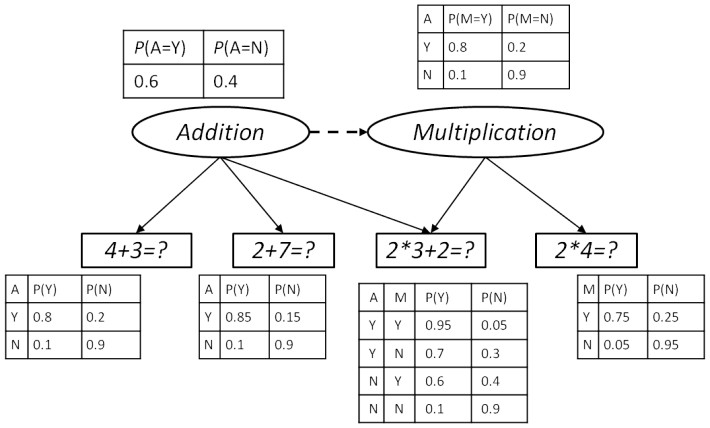


Figure 6.1: A hypothetical Bayesian network. Solid edges are given by item to skill mapping, dashed edges between skill variables are to be discovered from data. The conditional probability tables are to be learned.

relationships as the well-studied machine learning problem of learning a Bayesian network from data with the presence of unobserved latent variables. We represent the prerequisite structure using Bayesian networks that use latent binary variables to represent the student knowledge of a skill (i.e., mastery or not mastery), and observed binary variables that represent the student performance answering items (i.e., correct or incorrect).

Algorithm 6.1 describes the COMMAND pipeline. The input to COMMAND is a matrix $\mathbf{D}$ with $n \times p$ dimensions, representing n students, answering p items. Each entry in $\mathbf{D}$ encodes the performance of a student (see Table 6.1 for an example). Additionally, we require a Q-matrix to represent the item-to-skill mapping. Q-matrices are often designed by subject matter experts but automatic methods to discover them exist (González-Brenes, 2015).

COMMAND relies on a popular machine learning algorithm called *Structural Expectation Maximization* (Structural EM), which to the extent of our knowledge has not been used in educational applications before. Structural EM extends the Expectation Maximization (EM) algorithm to allow efficient structure learning of Bayesian networks when there are latent variables or missing values in the data. A secondary contribution of our work is introducing Structural EM for learning Bayesian network structures from educational data. We now describe the steps of COMMAND in detail.

Table 6.1: Example student performance matrix to use with COMMAND. The performance of a student is encoded with 1 if the student answered correctly the item, and 0 otherwise.

User	Item 1	Item 2	Item 3	Item p
Alice	0	1		0
Bob	1	1	...	1
Carol	0	0		1
		...		

Algorithm 6.1 The COMMAND algorithm

Require: A matrix $\mathbf{D}$ of student performance on a set of test items, skill-to-item mapping Q (containing a set of skills $\mathbf{S}$).

1: $G_0 \leftarrow \text{Initialize}(\mathbf{S}, Q)$ } Initialization
2: $i \leftarrow 0$ }
3: **do** } Structural EM
4: E-step:
5: $\Theta_i^* \leftarrow \text{ParametricEM}(G_i, \mathbf{D})$
6: $\mathbf{D}_i^* \leftarrow \text{Inference}(G_i, \Theta_i^*, \mathbf{D})$
7: M-step:
8: $\langle G_{i+1}, \Theta_{i+1} \rangle \leftarrow \text{BNLearning}(G_i, \mathbf{D}_i^*)$
9: $i \leftarrow i + 1$
10: **while** stop criterion is not met }
11: $RE \leftarrow FindReversibleEdges(G_i)$ } Discriminate between equivalent BNs
12: $EC \leftarrow EnumEquivalentDAGs(G_i)$
13: $DE \leftarrow \{\}$
14: **for** every reversible edge $S_i - S_j$ in RE **do**
15: $ratio \leftarrow \frac{P(S_j=0|S_i=0)}{P(S_i=0|S_j=0)}$ [1]
16: **if** $ratio \geq 1$ **then**
17: $ratio^* = ratio$
18: $DE \leftarrow DE \cup S_i \rightarrow S_j$
19: **else**
20: $ratio^* = \frac{1}{ratio}$
21: $DE \leftarrow DE \cup S_i \leftarrow S_j$
22: **end if**
23: **end for**
24: $sort(DE)$ by $ratio^*$ in descending order
25: **while** DE is not empty **do**
26: $e \leftarrow dequeue(DE)$
27: **if** $\exists G \in EC \; e \in G$ **then**
28: $\forall G \in EC$, remove G from EC if $e \notin G$
29: **end if**
30: **end while**
31: return EC

6.3.1 Initial Bayesian Network

COMMAND first creates an initial Bayesian network using the Q-matrix by creating an arc to each item from each of its required skills. Because there are no edges between the skills, this initial network does not encode any prerequisite information. COMMAND uses Structural EM to learn arcs (prerequisites) between the skill variables.

6.3.2 Structural EM

A common solution to learning a Bayesian network from data is the score-and-search approach (Cooper and Herskovits, 1992; Heckerman et al., 1997). This approach uses a scoring function (like the Bayesian Information Criterion (BIC)) to measure the fitness of a Bayesian network structure to the observed data, and it attempts to find the optimal model in the space of all possible Bayesian network structures. However, the conventional score-and-search approaches rely on efficient computation of the scoring function, which is only feasible for problems where data contain observations for all variables in the Bayesian network. Unfortunately, our domain has skill variables that are not directly observed. An intuitive work-around is to use EM to estimate the scoring function. However, in this case EM takes a large number (hundreds) of iterations that require Bayesian network inference, which is computationally prohibitive. Further, we need run EM for each candidate structure, and the number of possible Bayesian network structures is super-exponential with respect to the number of nodes. The Structural EM algorithm (Friedman, 1997) is an efficient alternative.

Structural EM is an iterative algorithm that inputs a matrix $\mathbf{D}$ of student performance (see example Table 6.1). Figure 6.2 illustrates one iteration of the Structural EM algorithm. The relevant steps are also sketched in Algorithm 6.1. Each iteration consists of an Expectation step (*E-step*) and a Maximization step (*M-step*). In the *E-step*, it first finds the maximum likelihood estimate Θ^* of the CPTs for the current structure G calculated from previous iteration using parametric EM. It then does Bayesian inference to compute the expected values for the latent variables using the current model (G, Θ^*), and uses the values to complete the data. In the

[1] $P(S_i = a|S_j = b)$ can be computed using any Bayesian network inference algorithm such as Junction tree algorithm (Koller and Friedman, 2009).

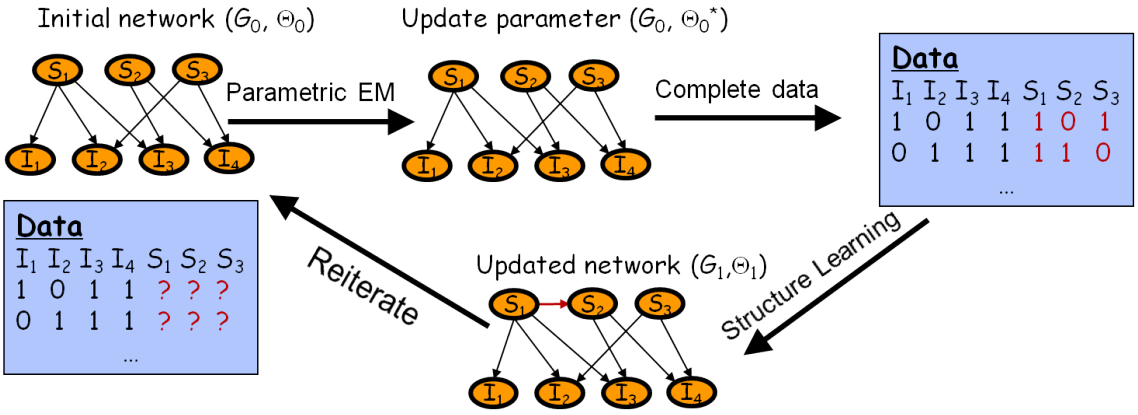


Figure 6.2: An illustration of the Structure EM algorithm to discover the structure of the latent variables. G represents the DAG structure. Θ is the set of conditional probability tables (CPTs).

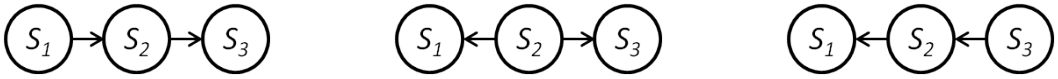
M-step, it uses the conventional score-and-search approach to optimize the structure according to the completed data (as if the latent variables were observed). Since the space of possible Bayesian network structures is super-exponential, exhaustive search is intractable and local search algorithms, such as greedy hill-climbing search, are often used. The *E-step* and *M-step* interleave and iterate until some stop criterion is met, e.g., the scoring function does not change significantly. Contrast to the conventional score-and-search algorithm, Structural EM runs EM only on one structure in each iteration, thus is computationally more efficient.

We use an efficient implementation of Structural EM available online called LibB[2]. Because COMMAND's initialization step fixes the arcs from skills to items according to the Q-matrix, the *M-step* only needs to consider the candidate structures that comply with the Q-matrix. An advantage of using Structural EM to discover the prerequisite relationship of skills is that it can be easily extended to incorporate domain knowledge. For example, we can place constraints on the output structure to force or to disallow a skill to be a prerequisite of another skill. Another advantage of Structural EM is that it can be applied when there are missing data in the student performance matrix $\mathbf{D}$ (Friedman, 1997). That is, some students do not answer all the items. The general idea is, in the *E-step*, the algorithm also computes the expected values for missing data points, in addition for latent variables.

[2] http://compbio.cs.huji.ac.il/LibB/programs.html

6.3.3 Discriminate Between Equivalent Bayesian Networks

Structural EM selects a Bayesian network model based on how well it explains the distribution of the data. Bayesian network theory states that some Bayesian networks are statistically equivalent in representing the data. Thus, the output from Structural EM is actually an equivalence class (EC) that may contain many Bayesian network structures[3]. These equivalent Bayesian networks have the same skeleton and the same v-structures[4]. For instance, Figure 6.3 gives an example of a simple equivalence class containing three Bayesian networks that are not distinguishable by Structural EM algorithm and the method in (Scheines et al., 2014). They share the skeleton but differ in the orientation of at least one of the edges (we will call such an edge a reversible edge). They apparently represent three different prerequisite structures.



(a) (b) (c)

Figure 6.3: Three equivalent Bayesian networks representing different prerequisite structures.

6.3.3.1 Domain Knowledge

To determine a unique structure, we use a heuristic based in domain knowledge to determine the orientation of each reversible edge. For convenience in notation, let's assume that the random variables that represent skill proficiency can take two values: 0 if the skills is not mastered, and 1 if the skill is mastered. Our assumption is that if a skill S_1 is the prerequisite of a skill S_2, a student can not master skill S_2 before she masters S_1. More formally:

Assumption 6.1. *If S_1 is a prerequisite of S_2 (i.e., $S_1 \rightarrow S_2$), then $S_1 = 0 \Rightarrow S_2 = 0$. In other words, $P(S_2 = 0|S_1 = 0) = 1$.*

Our assumption implies that S_1 cannot be a prerequisite of S_2 if $P(S_2 = 0|S_1 = 0) = 1$ does not hold. This puts a constraint on the joint distribution encoded by the Bayesian network to be learned.

[3]Structural EM outputs a DAG. However, the scoring function does not discriminate between the many DAGs of the equivalence class.

[4]A v-structure with nodes u, v, w in a DAG are the directed edges $u \rightarrow v$ and $w \rightarrow v$ and u and w are not adjacent in the DAG (Verma and Pearl, 1990).

For example, consider the case of choosing the orientation of a reversible edge $S_1 - S_2$ from $S_1 \leftarrow S_2$ or $S_1 \rightarrow S_2$. We can check whether $P(S_2 = 0|S_1 = 0) = 1$ or $P(S_1 = 0|S_2 = 0) = 1$. However, it is possible that our assumption does not hold, and a student got to master a skill even if he does not know the prerequisite. Moreover, because of statistical noise, the conditional probability $P(S_2 = 0|S_1 = 0)$ may not be exactly 1. Thus, we use the following empirical rule:

Rule 6.1. *If $P(S_2 = 0|S_1 = 0) \geq P(S_1 = 0|S_2 = 0)$, we determine $S_1 \rightarrow S_2$; otherwise, we determine $S_1 \leftarrow S_2$.*

Note that these two conditional probabilities can be computed easily from the Bayesian network model output from Structural EM. The intuition behind this rule is that the conditional probability $P(S_2 = 0|S_1 = 0)$ can be interpreted as the strength of the prerequisite relationship $S_1 \rightarrow S_2$. The larger of this probability, the more likely the relationship $S_1 \rightarrow S_2$ holds. Since here we are concerned with which direction the edge goes, we simply compare the two probabilities and select the direction that is more probable. Note that $P(S_2 = 0|S_1 = 0) = 1$ and $P(S_1 = 0|S_2 = 0) = 1$ may hold simultaneously. If $S_1 \rightarrow S_2$ is true, $P(S_1 = 0|S_2 = 0) = 1$ only if $P(S_1 = 1) = 0$ or if $P(S_2 = 0|S_1 = 1) = 0$.[5] If $P(S_1 = 1) = 0$, this implies that no student knows S_1. If $P(S_2 = 0|S_1 = 1) = 0$, it means that learning S_2 becomes trivial once students know S_1. For simplicity, we ignore this extreme case.

6.3.3.2 Theoretical Justification of Heuristic

We now provide theoretical justification for the rule we propose. Consider a simple equivalence class, which contains two equivalent DAGs $S_1 \rightarrow S_2$ and $S_1 \leftarrow S_2$, where the true model is $S_1 \rightarrow S_2$. We have three free conditional probability parameters: $P(S_1 = 0) = p$, $P(S_2 = 0|S_1 = 0) = q$, $P(S_2 = 1|S_1 = 1) = r$. Let's define a *ratio* that quantifies choosing the true model:

$$ratio = \frac{P(S_2 = 0|S_1 = 0)}{P(S_1 = 0|S_2 = 0)}. \tag{6.1}$$

Using Bayes rule and rules of probability, the rule $ratio \geq 1$ becomes $(1-p)(1-r)-p(1-q) \geq 0$. Since *ratio* depends on p, q and r, we study how *ratio* changes with these parameters. Figure 6.4

[5] Since $P(S_1 = 0|S_2 = 0) = \frac{P(S_2=0|S_1=0)P(S_1=0)}{P(S_2=0|S_1=0)P(S_1=0)+P(S_2=0|S_1=1)P(S_1=1)}$, $P(S_1 = 0|S_2 = 0) = 1$ only if $P(S_2 = 0|S_1 = 1)P(S_1 = 1) = 0$.

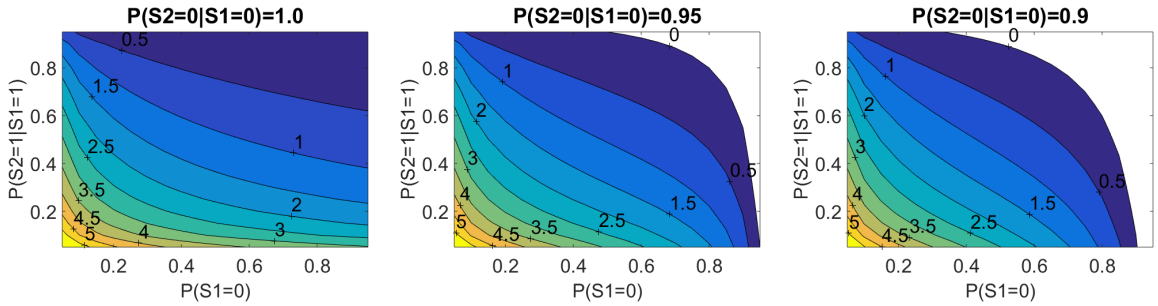


Figure 6.4: Contour plots of $log(ratio)$ against $P(S_1 = 0)$ and $P(S_2 = 1|S_1 = 1)$ for various values of $P(S_2 = 0|S_1 = 0)$.

shows the contour plots of $log(ratio)$ against $P(S_1 = 0)$ and $P(S_2 = 1|S_1 = 1)$ for three different values of $P(S_2 = 0|S_1 = 0)$. The white region in each contour plot is the region where our heuristic fails because $ratio < 1$. Figure 6.4(a) shows that when $P(S_2 = 0|S_1 = 0) = q = 1$, our heuristic rule is always correct, no matter what, because there is no white space. With $P(S_2 = 0|S_1 = 0)$ decreasing, the white region becomes larger and the rule becomes less accurate. As mentioned, $P(S_2 = 0|S_1 = 0)$ can be interpreted as the strength of the prerequisite relationship. If we fix the value of $P(S_2 = 0|S_1 = 0)$ and assume that the two free parameters p and r are independent and uniformly distributed, then the area of the white region represents the probability that the rule makes a wrong decision. As the strength of the prerequisite relationship gets weaker, our rule to determine the prerequisite relationship becomes less accurate.

6.3.3.3 Orient All Reversible Edges

Using our proposed rule, we can orient every reversible edge in the network structure. However, orienting each reversible edge is not independent and may conflict with each other. Having oriented one edge would constrain the orientation of other reversible edges because we have to ensure the graph is a DAG and the equivalence property is not violated. For example, in Figure 6.5a, if we have determined $S_1 \rightarrow S_2$, the edge $S_2 \rightarrow S_3$ is enforced. In this paper, we take an ad-hoc strategy to determine the orientation for all reversible edges. For each reversible edge $S_i - S_j$, we let $ratio^* = ratio$ if $ratio \geq 1$ and $ratio^* = \frac{1}{ratio}$ otherwise. The larger the $ratio^*$

is, the more confidently when we decide the orientation. We sort the list of reversible edges by $ratio^*$ in descending order. We then orient the edges by this ordering. In our implementation, we use the following strategy: we first enumerate all equivalent Bayesian networks and make them a list of candidates; when an edge is oriented to $S_i \rightarrow S_j$, we remove all contradicting Bayesian networks from the list. Eventually only one Bayesian network structure stands. This procedure is detailed in the *Discriminate between equivalent BNs* section of Algorithm 6.1. The $EnumEquivalentDAGs(G_i)$ implements the algorithm of enumerating equivalent DAGs in Algorithm 3.2.

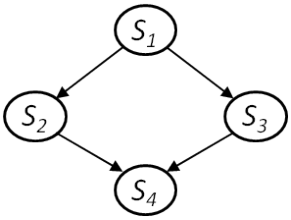
6.4 Evaluation

In § 6.4.1, we evaluate COMMAND with simulated data to assess the quality of the discovered prerequisite structures. Then, in § 6.4.2 we use data collected from real students. In all our experiments, we use BIC as the scoring function in Structural EM .

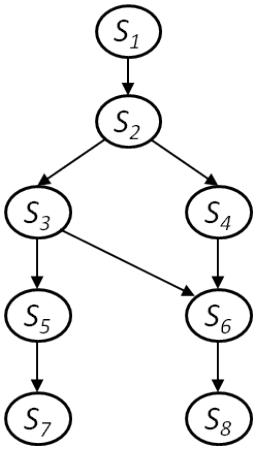
6.4.1 Simulated Data

Synthetic data allow us to study how COMMAND compares to the ground truth. For this, we engineered three prerequisite structures (DAGs), shown in Figure 6.5. Here, each figure represents different causal relations between the simulated latent skill variables.

(a) Structure 1



(b) Structure 2



(c) Structure 3

Figure 6.5: Three different DAGs between latent skill variables. Item nodes are omitted.

For clarity, Figure 6.5 omits the item nodes; but each skill node is parent of six item variables and each item variable has 1-3 skill nodes as parents. All of these nodes are modeled using binary random variables. More precisely, the latent nodes represent whether the student achieves mastery of the skill, and the observed nodes indicate if the student answers the item correctly. Notice that these networks include the prerequisite structures as well as the skill-item mapping.

We consider simulated data with different number of observations ($n = 150, 500, 1000, 2000$). For each sample size and each DAG, we generate ten different sets of conditional probability tables randomly with three constraints. First, we enforce that achieving mastery of the prerequisites of a skill will increase the likelihood of mastering the skill. Second, for each prerequisite pair $S_i \rightarrow S_j$, $P(S_j = 0|S_i = 0)$ is randomly selected to be in $[0.9, 1.0]$. Finally, mastery of a skill increases the probability of student correctly answering the test item. In total we generated 120 synthetic datasets (3 DAGs x 4 sample sizes x 10 CPTs), and report the average results.

We evaluate how well COMMAND can discover the true prerequisite structure using metrics designed to evaluate Bayesian networks structure discovery. In particular, we use the F_1 *adjacency score* and the F_1 *orientation score*. The adjacency score measures how well we can recover connections between nodes. It is a weighted average of the true positive adjacency rate and the true discovery adjacency rate. On the other hand, the orientation score measures how well we can recover the direction of the edges. It is calculated as a weighted average of the true positive orientation rate and true discovery orientation rate. In both cases, the F_1 score reaches its best value at 1 and worst at 0. Moreover, for comparison, we compute the F_1 *adjacency score* for Bayesian network structures whose skill nodes are fully connected with each other. These fully connected DAGs will serve as baselines for evaluating the adjacency discovery[6]. For completeness, we list these formulas in Tables 6.2 and 6.3, respectively.

We use these metrics to evaluate the effect of varying the number of observations of the training set (sample size) on the quality of learning the prerequisite structure. We designed

[6]We do not compute F_1 orientation score for fully connected DAGs because all edges in a fully connected DAG are reversible.

Table 6.2: Formulas for measuring adjacency rate (AR)

Metric	Formula
True positive ($TPAR$)	$\frac{\text{\# of correct adjacencies in learned model}}{\text{\# of adjacencies in true model}}$
True discovery ($TDAR$)	$\frac{\text{\# of correct adjacencies in learned model}}{\text{\# of adjacencies in learned model}}$
F_1-AR	$\frac{2 \cdot TPAR \cdot TDAR}{TPAR + TDAR}$

Table 6.3: Formulas for measuring orientation rate (OR)

Metric	Formula
True positive ($TPOR$)	$\frac{\text{\# of correctly directed edges in learned model}}{\text{\# of directed edges in true model}}$
True discovery ($TDOR$)	$\frac{\text{\# of correctly directed edges in learned model}}{\text{\# of directed edges in learned model}}$
F_1-OR	$\frac{2 \cdot TPOR \cdot TDOR}{TPOR + TDOR}$

experiments to specifically answer the following four questions:

1. How does the type of items affect COMMAND's ability to recover the prerequisite structure? We consider the situation where in the model each item requires only one skill and the situation where each item requires multiple skills.

2. How well does COMMAND perform when there is noise in the data? We focus on studying noise due to the presence of unaccounted latent variables.

3. How well does COMMAND perform when the student performance data have missing values?

4. How is COMMAND compared with other prerequisite discovery methods? In particular, we compare COMMAND to the Probabilistic Association Rules Mining (PARM) method (Chen et al., 2015).

We now investigate these questions.

6.4.1.1 Single-skill vs Multi-skill Items

We consider two situations where different types of Q-matrix are used. In the first situation, each item node maps to exactly one skill node. In the second one, each item maps to 1-3 skills.

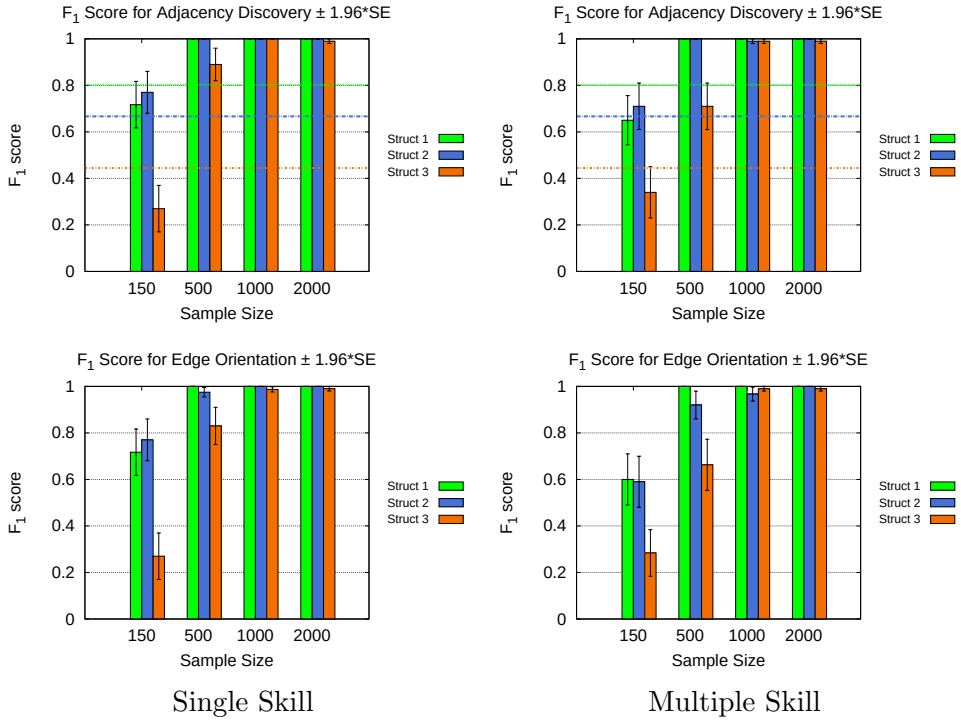


Figure 6.6: Comparison of F_1 scores for adjacency discovery (top row) and for edge orientation (bottom row). Horizontal lines are baseline scores for fully-connected (complete) networks. The error bars show the 95% confidence intervals, i.e., $\pm 1.96 * \texttt{SE}$.

Figure 6.6 compares the F_1 of adjacency discovery and edge orientation results under the two types of Q-matrices. With only 500 observations, COMMAND improves on a fully connected Bayesian network baseline. COMMAND's accuracy improves with the amount of data, but its accuracy is slightly lower when the Q-matrix contains items that require more than one skill. A possible explanation for this is that multi-skill items may introduce more spurious correlations in the data. With just 2000 observations, COMMAND recovers the true structures almost perfectly.

6.4.1.2 Sensitivity to Noise

Real-world data sets often contain various types of noise. For example, noise may occur due to latent variables that are not explicitly modeled. To evaluate the sensitivity of COMMAND to noise, we synthesize the three Bayesian networks in Figure 6.5 to include a *StudentAbility*

node that takes three possible states (low/med/high). In these Bayesian networks, students' performance depends not only on whether they have mastered the skills, but also on their individual ability. For simplicity, all items in the setting are single-skilled items. We first simulated data from Bayesian networks that have a *StudentAbility* variable to generate "noisy" data samples, and Figure 6.7 illustrates the procedure of this sensitivity analysis experiment for Structure 1.

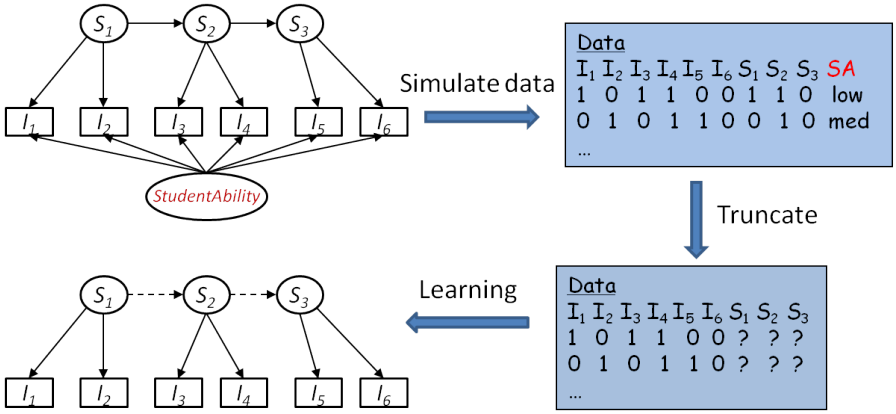


Figure 6.7: Evaluation of COMMAND with noisy data.

Figure 6.8 compares the results where noise was introduced or not. Interestingly, the noise actually improves COMMAND's accuracy. This improvement is more evident when the sample size is small (see $n = 150$). For smaller sample sizes, Structural EM usually discovers less relationships than actually exist, because BIC prefers sparse structures. We hypothesize that the correlations caused by *StudentAbility* node would cause Structural EM to add "stronger" edges between skill nodes, resulting in higher F_1.

6.4.1.3 Sensitivity to Missing Values

Real-world datasets collected from students often have missing values, for example, when learners do not answer all items. To evaluate how COMMAND performs on data with missing values, we generated data sets of with 1000 observations with varying fraction of randomly missing values (10%, 20%, 30%, 40%, 50%). We used COMMAND to recover the structures from these data sets. Again, the models only contain single-skilled items. Figure 6.9 shows the

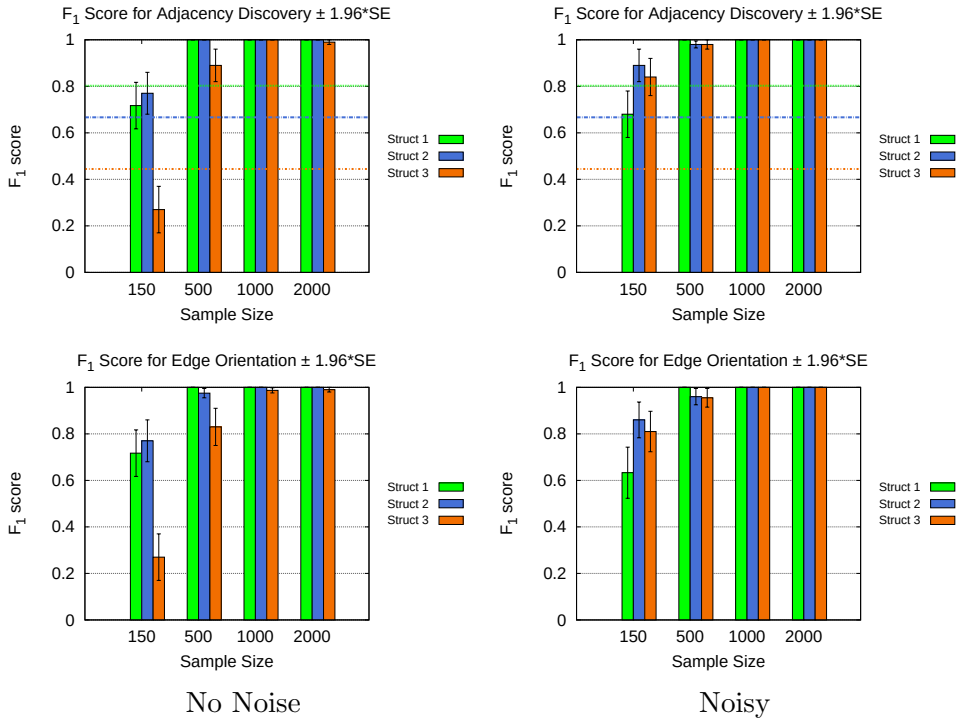


Figure 6.8: Results of adding systematic noise. Top: Comparison of F_1 scores for adjacency discovery. Horizontal lines are baseline F_1 scores computed for fully connected Bayesian networks. Bottom: Comparison of F_1 scores for edge orientation.

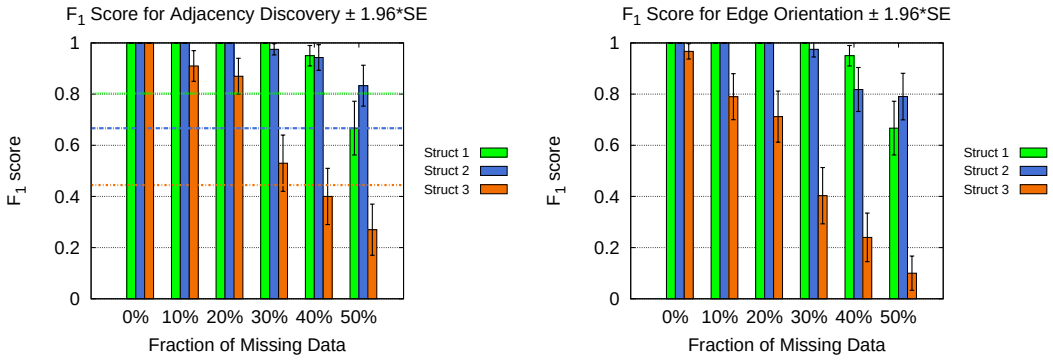


Figure 6.9: Results of learning with missing data. Left: Comparison of F_1 scores for adjacency discovery. Horizontal lines are baseline F_1 scores computed for fully connected Bayesian networks. Right: Comparison of F_1 scores for edge orientation.

results of this experiment. Although accuracy decreases when the fraction of missing values increases, COMMAND is able to recover the true structures for Structure 1 and 2 even when the data contain up to 30% missing values.

6.4.1.4 Comparison With Prior Work

The Probabilistic Association Rules Mining (PARM) is a recent algorithm for discovering the prerequisite relationships between skills (Chen et al., 2015). In this approach, a prerequisite relationship $S_1 \rightarrow S_2$ is considered to exist if $P(S_1 = 1, S_2 = 1) \geq minsup \wedge P(S_1 = 1|S_2 = 1) \geq minconf) \geq minprob$ and $P(P(S_1 = 0, S_2 = 0) \geq minsup \wedge P(S_2 = 0|S_1 = 0) \geq minconf) \geq minprob$, where $minsup$, $minconf$ and $minprob$ are pre-specified constants between 0 and 1.

We simulate data from Structure 3 from Figure 6.5(c) (with single-skilled items), which has 21 pair-wise prerequisite relationships. We derive pair-wise prerequisite relationships from this network and see how the two approaches discover these relationships. When experimenting with PARM, we use $minsup = 0.125$, $minconf = 0.76$, $minprob = 0.9$, because they were suggested by the authors (Chen et al., 2015).

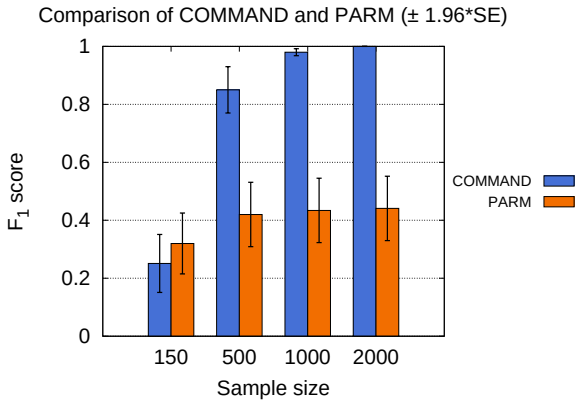


Figure 6.10: Comparison of COMMAND and PARM for discovering prerequisite relationships in Structure 3.

PARM is limited to discovering pair-wise prerequisite relationships (instead of constructing the full structure). To make a fair comparison, we evaluate how accurately COMMAND and PARM discover relationship pairs. For this, we use the F1 metric in Table 6.2, but we count

pairs of *related skills* instead of adjacencies. Two skills are related if one is a descendant of the other one. Figure 6.10 shows that COMMAND outperforms PARM, and the difference becomes significant for sample size $n \geq 500$. The low F_1 score of by PARM is because it fails to discover many prerequisite relationships (data not shown), and because PARM does not respect transitivity. For example, PARM may reject $S_1 \rightarrow S_3$ even it has discovered $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_3$. We speculate that selecting a different set of cutoff values for PARM may improve the results. However, determining these thresholds is not trivial and may require experts' intervention. By contrast, COMMAND does not require tuning.

6.4.2 Real Student Performance Data

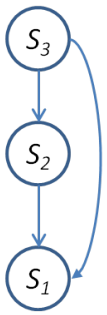
We now evaluate COMMAND using two real-world data sets.

6.4.2.1 English Data Set

The Examination for the Certification of Proficiency in English (ECPE) dataset describes 2922 examines in their understanding of English language grammar (Templin and Bradshaw, 2014). The dataset includes student performance in 28 items on 3 skills (S_1: morphosyntactic rules, S_2: cohesive rules, and S_3:lexical rules). Each item requires either one or two of the three skills.

$P(S_3=0)$	$P(S_3=1)$
0.33	0.67

S_3	$P(S_2=0 \mid S_3)$	$P(S_2=1 \mid S_3)$
0	0.90	0.10
1	0.20	0.80



S_2	S_3	$P(S_1=0 \mid S_2,S_3)$	$P(S_1=1 \mid S_2,S_3)$
0	0	0.97	0.03
1	0	0.57	0.43
0	1	0.87	0.13
1	1	0.33	0.67

Figure 6.11: The estimated DAG and CPTs of the ECPE data set.

Figure 6.11 shows the prerequisite structure discovered with COMMAND. It hypothesizes that lexical rules is a prerequisite of cohesive rules and morphosyntactic rules; cohesive rules

is a necessary skill for learning morphosyntactic rules. The pair-wise prerequisite relationships totally agree with the findings in (Templin and Bradshaw, 2014) and that by the PARM method in (Chen et al., 2015). Our model infers a complete DAG, suggesting that there are no conditional independencies among the three skills. This is an interesting insight that previous approaches cannot provide. Further, COMMAND also outputs the conditional probabilities associated with each skill and its direct prerequisites. We clearly see that the probability of student mastering a skill increases when the student has acquired more prerequisites of the skill.

6.4.2.2 Math Data Set

We now evaluate COMMAND using data collected from a commercial non-adaptive tutoring system. The textbook items are classified in chapters, sections, and objectives. We only use student performance data from tests in Chapter 2 and 3. That is, students are tested on the items after they have been taught all relevant skills.

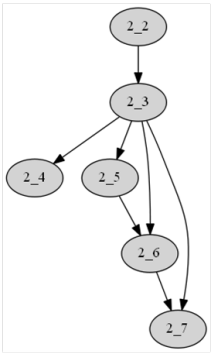
Q-matrix and preprocessing We define skills as book sections. We use a Q-matrix that assigns each exercise to a skill solely as the book section in which the item appears.[7] For each chapter, we process the data to find a subset of items and students that do not have missing values. That is, the datasets we use in COMMAND have students responding to *all* of the items.

After filtering, two data sets, `Math-chap2` and `Math-chap3`, were obtained for Chapter 2 and 3 respectively. In `Math-chap2`, six skills are included and each skill is tested on three to eight items, for a total of 30 items. In `Math-chap3`, seven skills are included and each skill has three to seven items, for a total of 33 items. `Math-chap2` includes student test results for 1720 students, while the `Math-chap3` has test results for 1245 students. For simplicity we use binary variables to encode performance data and skill variables.

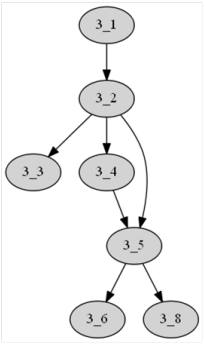
Prerequisite Structure Discovery The Bayesian networks generated with the COMMAND algorithm are illustrated in Figure 6.12. Our observation is that the topological order

[7] Here we assume the items are single-skilled despite that they might be multi-skilled.

Skill ID	Skill Name
2_2	Symbols and Sets of Numbers
2_3	Fractions and Mixed Numbers
2_4	Exponents, Order of Operations, Variable Expressions, and Equations
2_5	Adding Real Numbers
2_6	Subtracting Real Numbers
2_7	Multiplying and Dividing Real Numbers



(a) Prerequisite structure learned for `Math-chap2`



Skill ID	Skill Name
3_1	Simplifying Algebraic Expressions
3_2	The Addition and Multiplication Properties of Equality
3_3	Solving Linear Equations
3_4	An Introduction to Problem Solving
3_5	Formulas and Problem Solving
3_6	Percent and Mixture Problem Solving
3_8	Solving Linear Inequalities

(b) Prerequisite structure learned for `Math-chap3`

Figure 6.12: Prerequisite structures constructed by COMMAND for `Math` data sets.

of the sections in both structures are fully consistent with the book ordering heuristic. This shows an agreement between our fully data-driven method and human experts. We also ran PARM approach to learn pair-wise prerequisite relationships from these data sets. Given $minsup = 0.125$, $minconf = 0.76$ and $minprob = 0.9$, 2_5 $\rightarrow$ 2_6, 2_5 $\rightarrow$ 2_7 and 2_6 $\rightarrow$ 2_7 are discovered for `Math-chap2`, 3_1 $\rightarrow$ 3_3 and 3_2 $\rightarrow$ 3_3 are discovered for `Math-chap3`. These relationships are small subset of the set of relationships discovered by COMMAND.

Predictive Performance COMMAND outputs a Bayesian network model that can be used for inference and predictive modeling. For example, given a student's response to a set of items, we can infer the student's knowledge status of a skill. We could use COMMAND to identify students that may need remediation because they lack some background. We evaluate the accuracy of the predicted student performance on an item, when we observe the student response on the other items. More precisely, we compute the posterior probability of a student's response to an item I_i given his performance on all other items $\mathbf{I}_{-i} = \mathbf{I} \setminus \{I_i\}$, by marginalizing over the set of latent skill variables $\mathbf{S}$:

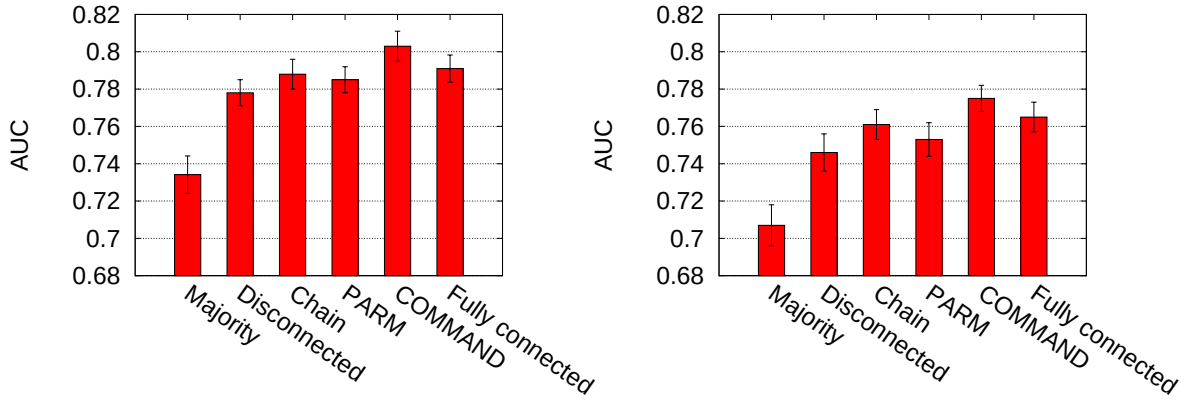
$$P(I_i|\mathbf{I}_{-i} = \mathbf{i}_{-i}) = \sum_{\mathbf{S}} P(I_i, \mathbf{S}|\mathbf{I}_{-i} = \mathbf{i}_{-i}).$$

This probability can be computed efficiently using the Junction tree algorithm (Koller and Friedman, 2009). We then do binary classification based on the posterior probability to determine if the student is likely to answer correct. We compare the Bayesian network models

generated from COMMAND with five baseline predictors:

- A *majority* classifier which always classifies an instance to the majority class. For example, if majority of the students get an item wrong, other students would likely get it wrong.
- A Bayesian network model in which the skill variables are *disconnected*. This model assumes that the skill variables are marginally independent of each other. Most existing knowledge tracing approaches make this assumption.
- A Bayesian network model in which the skill variables are connected in a *chain* structure, i.e., 2-2→2-3→2-4→ . . . . This assumes that a section (skill) only depends on the previous section. In other words, a first-order Markov chain dependency structure.
- A Bayesian network model constructed using the pairwise relationships output from *PARM*. That is, we create an edge $S_i \rightarrow S_j$ if PARM says S_i is the prerequisite of S_j.
- A *fully connected* Bayesian network where skill variables are fully connected with each other. This model assumes no conditional independence between skill variables and can encode any joint distribution over the skill variables. However, it has exponential number of free parameters and thus can easily overfit the data.

The parameters of these baseline Bayesian network predictors are estimated from the data using parametric EM. The model predictions were evaluated using the *Area Under the Curve* (AUC) of the Receiver Operating Characteristic (ROC) curve metric calculated from 10-fold cross-validation. Results are presented in Figure 6.13. The error bars show the 95% confidence intervals calculated from the cross-validation. On both `Math-chap2` and `Math-chap3` data sets, the COMMAND models outperform the other five models. The *fully connected* models are the second best performing models. On `Math-chap2`, COMMAND model has an AUC of 0.803 ± 0.008 and the *fully-connected* model has an AUC of 0.791 ± 0.007 (Figure 6.13a). A paired t-test reveals that the AUC difference of two models are statistically significant with a p-value of 0.0022. On `Math-chap3`, COMMAND model has an AUC of 0.775 ± 0.007 and the



(a) `Math-chap2` AUC results (b) `Math-chap3` AUC results

Figure 6.13: Ten fold cross-validation results of evaluating the predictions of student performance.

fully-connected model has an AUC of 0.765 ± 0.008 (Figure 6.13b). The AUC difference of two models are also statistically significant with a p-value of 0.01. The *fully connected* models are outperformed by the much simpler prerequisite models, suggesting overfitting.

6.5 Conclusion and Discussion

Prerequisite graphs have been shown (Botelho et al., 2015; Käser et al., 2014) to improve student models. However, discovering the prerequisites between skills requires significant effort from subject matter experts. The main contribution of our work is a novel algorithm that simultaneously infers a prerequisite graph and a student model from data with less human intervention.

We extend on prior work in significant ways. We optimize the full structure of skills that captures the conditional independence between skills, instead of only estimating the pairwise relationships. Our experiments suggests that this results in better accuracy. Moreover, we argue that our strategy is easier to use because it does not require manual tuning of parameters. Other methods (Brunskill, 2010) require the *guess* and *slip* probabilities to be provided as input, or alternatively (Chen et al., 2015), thresholds to determine the existence of a prerequisite relationship. Determining these values requires experts' intervention. COMMAND does not require such tuning.

We analyze how missing values, noise and dataset size can affect the performance of COMMAND. Further research could explore additional datasets and baselines. A secondary contribution of our work is that we develop a methodology to evaluate prerequisite structures on real student data. We believe that we are the first to compare prerequisite discovery strategies by how well they can be used to predict student performance. Therefore, we validate COMMAND not only with synthetic data, but with two real-world datasets. Our results suggest that COMMAND improves on the state of the art because it significantly improves on a recently published technique.

Learning a prerequisite graph is not merely discovering a Bayesian network— equivalent Bayesian network structures in fact represent different prerequisite structures. We believe we are the first to address this problem. We use domain knowledge to refine the prerequisite models output from the Bayesian network structure learning algorithms using a theoretically motivated method.

CHAPTER 7. SUMMARY, CONTRIBUTIONS AND FUTURE WORK

In this chapter, we summarize the main contributions of this thesis and discuss the potential future work related to structure discovery in Bayesian networks.

7.1 Summary and Contributions

Summary of Chapter 1

We introduced the syntax and semantics of Bayesian networks and defined the research problems of Bayesian network structure learning and structure discovery. We then reviewed related work and briefly discussed their pros and cons. Finally, we provided an overview of this thesis.

Summary and Contributions of Chapter 2

We studied the problem of learning a Bayesian network structure from the data and proposed a novel heuristic algorithm that takes advantage of the idea of curriculum learning and learns Bayesian network structures by stages. At each stage a subnet is learned over a selected subset of the random variables. The selected subset grows with stages and eventually includes all the variables. We proved theoretical advantages of our algorithm and also empirically showed that it outperformed the state-of-the-art heuristic approach in learning Bayesian network structures under several different evaluation metrics.

Summary and Contributions of Chapter 3

We developed an algorithm to efficiently enumerate the k-best equivalence classes of Bayesian networks where Bayesian networks in the same equivalence class are equally expressive in terms

of representing probability distributions. Our algorithm is capable of finding much more best DAGs than the previous algorithm that directly finds the k-best DAGs (Tian et al., 2010). We demonstrated our algorithm on the task of Bayesian model averaging for computing the posterior probabilities of edge features. Our approach goes beyond the maximum-a-posteriori (MAP) model by listing the most likely network structures and their relative likelihood and therefore has important applications in causal structure discovery.

Summary and Contributions of Chapter 4

We presented a parallel algorithm for exact Bayesian edge learning. Our algorithm computes the exact posterior probabilities for all possible directed edges with optimal space efficiency and nearly optimal time efficiency. This is the first practical parallel algorithm for computing the exact posterior probabilities of structural features in Bayesian networks. We demonstrated its capability on datasets with up to 33 variables and its scalability on up to 2048 processors. To our knowledge, 33-variable network is the largest problem solved so far. We demonstrated our algorithm on a biological data set for discovering the (*yeast*) pheromone response pathways. Further, we developed two parallel techniques for computing two variants of well-known *zeta transform*. These features or ideas can potentially be extended and applied in developing parallel algorithms for related problems.

Summary and Contributions of Chapter 5

We developed a novel algorithm to compute the exact posterior probabilities of ancestor relations in Bayesian networks. Previous Bayesian approach assumes an order-modular prior over DAGs and performs summation over order space instead of DAG space. As a result, the computed posteriors would bias towards DAGs consistent with more linear orders and the Markov equivalence is not respected either. Instead, our algorithm allows the uniform structure prior and respects the Markov equivalence by directly summing over DAG space. Experimental comparison showed that the structure prior has non-negligible effect on the computed posteriors. We also applied our algorithm on a biological data set to discover protein signaling pathways. Further, we extended our algorithm to compute the exact posterior of any $s \rightsquigarrow p \rightsquigarrow t$ relation,

i.e., a directed path from s to t via p.

Summary and Contributions of Chapter 6

We studied the problem of estimating the prerequisite relationships between skills from student performance data. We introduced *Combined student Modeling and prerequisite Discovery* (COMMAND), a novel algorithm for jointly inferring a skill prerequisite graph and a student model. COMMAND learns the prerequisite relations as a Bayesian network that allows modeling of the full prerequisite structure of skills. Our experiments on simulations and real student data suggested that COMMAND is more accurate than prior methods for prerequisite discovery and student modeling.

7.2 Future Work

Some interesting directions for future work include:

- In chapter 2, our algorithm incrementally constructs the Bayesian network structure by growing a subnet stage by stage. In each stage, the algorithm only learns the subnet over a selected subset of the variables. An alternative way is to use ideas from the bottom-up hierarchical clustering (HC). That is, we may cluster variables using HC based on mutual information and learn a subnet over each cluster. When merging two smaller clusters into one larger cluster, we learn a new subnet over the newly formed cluster using the subnets over the two smaller clusters as starting point. In other words, we simultaneously learn multiple components of the target structure at each time and join these components to make larger ones. A benefit of this algorithm is that it can be easily parallelized as learning of each component is independent and this will potentially make it more scalable.

- In chapter 3, *kBestEC* and *kBestDAG* are both based on the DP algorithm that finds an optimal Bayesian networks. Recently, integer linear programming (ILP) has been used to find the optimal Bayesian network and showed competitive or faster than the DP algorithm (Jaakkola et al., 2010; Cussens, 2011; Bartlett and Cussens, 2013). In particular, ILP based algorithm GOBNILP casts the structure learning problem as a

linear program. In such setting, each DAG can be encoded as a linear constraint (linear inequality). Thus, we can run GOBNILP to iteratively find the top k DAGs as follows: in the first iteration, we simply run GOBNILP to find the best DAG; the second best DAG can be learned by starting a new ILP program with a the linear constraint encoding the best DAG added to rule out the best DAG; the k-best DAGs can be found by iteratively running the IP search k times with appropriate constraints added in each time. That is, to find the i-th best DAG, an IP search is launched with $i-1$ linear constraints added to rule out the $(i-1)$-best DAGs. A more efficient way to find the k-best DAGs is to consider the equivalence between DAGs. That is, when find a DAG with ILP, we simply run Algorithm 3.2 to find all its equivalent DAGs. In the next iteration, we encode these DAGs into their corresponding linear constraints and add these constraints in the new ILP program to rule out the entire equivalence class (EC). In such way, we can efficiently find the k-best ECs. Alternatively, if we can directly find a linear representation for each equivalence class, finding the k-best ECs using GOBNILP will be much more efficient. Thus, one direction of the future work is to explore this idea.

- In chapter 4, from the experiments, we observed that the memory usage of our ParaREBEL reached the limit much faster than computing time did. Solving a 33-variable problem took less than one hour on a typical computing cluster with thousands of processors, but it required more than two terabytes of memory. Thus, one direction of the future work is to improve the algorithm such that less space is used. Particularly, there is a possibility to combine the present parallel algorithm with the method in (Parviainen and Koivisto, 2010) to trade space against time.

- In chapter 6, the first phase of COMMAND algorithm uses Structural EM to learn a Bayesian network with skills as latent variables. Structural EM interleaves greedy structure search with the estimation of latent variables and parameters, maintaining a single best network at each step. It does not provide any theoretical guarantee on the quality of the networks learned. Recently, Lazic et al. (2013) proposed Structural Expectation Propagation (SEP), an extension of EP that can also infer the structure of Bayesian networks

having latent variables and missing data. SEP accounts for uncertainty in structure and parameter values and returns a variational distribution over network structures rather than a single network. A MAP network can then be obtained based on the estimated distribution. Experimental comparison showed that SEP outperformed Structural EM in learning the DAG structure. Thus, one direction of the future work is to replace Structural EM with SEP for structure learning in the first phase of COMMAND. Further, in the second phase of COMMAND to identify a unique DAG from its equivalence class, we take an ad-hoc strategy to orient all reversible edges, one by one. This strategy is very sensitive to errors since an error made on one edge can result in propagated errors in next stages. Thus, another piece of future work is to design a metric that can measure the global fitness of a DAG to the domain constraint and use this metric to identify a unique DAG. In this way, learning may be more accurate and even more efficient.

APPENDIX A. SUPPLEMENTAL MATERIAL FOR FINDING THE K-BEST EQUIVALENCE CLASSES FOR MODEL AVERAGING

Section A.1 provides the proofs of theorems and lemmas in chapter 3. Section A.2 gives the detailed algorithm for checking equivalence of two DAGs used by Algorithm 3.1 in chapter 3.

A.1 Proofs of Theorems

Lemma 3.1. For any decomposable score function that satisfies score equivalence, we have $score(G_W) = score(G'_W)$ if G_W and G'_W are equivalent over node set $W \subseteq V$.

Proof. We can construct two equivalent DAGs G_V and G'_V over the total node set V from G_W and G'_W respectively. For each $v \in V \setminus W$, we pick arbitrary parent set $Pa_v \subseteq W$. Then G_V and G'_V can be constructed by $G_V = G_W \oplus_{v \in V \setminus W} Pa_v$ and $G'_V = G'_W \oplus_{v \in V \setminus W} Pa_v$. G_V and G'_V are equivalent since they have the same skeleton and same set of v-structures. This is because: G_W and G'_W have the same skeleton and the same set of v-structures; adding Pa_v's for all $v \in V \setminus W$ only adds directed edges from any node $u \in W$ to node $v \in V \setminus W$. This produces the same skeleton as well as the same set of v-structures. By Definition 3.2 and Definition 3.3,

$$score(G_V) = \sum_{v \in W} score_v(Pa_v^{G_W}) + \sum_{v \in V \setminus W} score_v(Pa_v) = score(G_W) + \sum_{v \in V \setminus W} score_v(Pa_v),$$

$$score(G'_V) = \sum_{v \in W} score_v(Pa_v^{G'_W}) + \sum_{v \in V \setminus W} score_v(Pa_v) score(G'_W) + \sum_{v \in V \setminus W} score_v(Pa_v),$$

$$score(G_V) = score(G'_V).$$

Solving these equations yields $score(G_W) = score(G'_W)$. □

Theorem 3.2. The k DAGs corresponding to the k-best scores output by the best-first search represent the k-best ECs over W with s as a sink.

Proof. We first prove that these k DAGs are mutually nonequivalent. For any $G^p_{W,s}$ and $G^q_{W,s}$, $p, q \in \{1, ..., k\}$, $p \neq q$, we have two different cases.

Case 1: $G^p_{W,s}$ and $G^q_{W,s}$ are constructed from $G^i_{W\setminus\{s\}}$ and $G^j_{W\setminus\{s\}}$ respectively. $G^1_{W\setminus\{s\}}$,..., $G^k_{W\setminus\{s\}}$ over $W \setminus \{s\}$ are nonequivalent. This implies that any two of them, say, $G^i_{W\setminus\{s\}}$ and $G^j_{W\setminus\{s\}}$, are either have different skeletons or have the same skeleton but different v-structures. Since adding parents Pa_s for s changes neither the skeleton nor any v-structures in $G^i_{W\setminus\{s\}}$ and $G^j_{W\setminus\{s\}}$, $G^p_{W,s}$ and $G^q_{W,s}$ must either have different skeletons or have the same skeleton but different v-structures. Therefore, $G^p_{W,s}$ and $G^q_{W,s}$ are not equivalent.

Case 2: $G^p_{W,s}$ and $G^q_{W,s}$ are constructed from the same $G^i_{W\setminus\{s\}}$ but with different parent sets for s. Since two different parent sets for s have different nodes, the sets of edges respectively added to $G^i_{W\setminus\{s\}}$ to construct $G^p_{W,s}$ and $G^q_{W,s}$ are different. As a result, $G^p_{W,s}$ and $G^q_{W,s}$ have different skeletons. Therefore, they are not equivalent.

Now we prove that the output $G^1_{W,s}, ..., G^k_{W,s}$ are the k-best over W with s as a sink. All we have to show is that for each equivalence class $EC^i_{W\setminus\{s\}}$, it is safe to keep just one DAG $G^i_{W\setminus\{s\}}$ while discarding others. That is, using another member $G'^i_{W\setminus\{s\}}$, we are unable to construct a DAG $G'_{W,s} = G'^i_{W\setminus\{s\}} \oplus Pa_s$ such that $score(G'_{W,s}) > score(G^k_{W,s})$ and it is nonequivalent to any of $G^1_{W,s}, ..., G^k_{W,s}$. Assume we can construct such $G'_{W,s}$, then we can construct an equivalent DAG by $G_{W,s} = G^i_{W\setminus\{s\}} \oplus Pa_s$. By Lemma 3.1, $score(G_{W,s}) = score(G'_{W,s}) > score(G^k_{W,s})$. Best-first search guarantees that this $G_{W,s}$ is in the list of $G^1_{W,s}, ..., G^k_{W,s}$. This contradicts the assumption that $G'_{W,s}$ is nonequivalent to any of $G^1_{W,s}, ..., G^k_{W,s}$.

Thus, Theorem 3.2 holds. □

A.2 Algorithms

$CheckEquivalence(G_W, G'_W)$ determines whether two DAGs G_W, G'_W over W are equivalent.

Algorithm A.1 *CheckEquivalence*(G_W, G'_W)

```
1: function CheckVStructure(v, G_W, G'_W)
2:     for each pair of distinct u, w ∈ Pa_v^W in G_W do
3:         if u ∉ Pa_w^W and w ∉ Pa_u^W and
4:           (u ∉ Pa_v^'W or w ∉ Pa_v^'W in G'_W) then
5:             return false
6:         end if
7:     end for
8:     for each pair of distinct u, w ∈ Pa_v^'W in G'_W do
9:         if u ∉ Pa_w^'W and w ∉ Pa_u^'W and
10:          (u ∉ Pa_v^W or w ∉ Pa_v^W in G_W) then
11:            return false
12:        end if
13:    end for
14:    return true
15: end function
16: /* Check skeleton */
17: for each node v ∈ W do
18:    for each u ∈ Pa_v^W in G_W do
19:        if u ∉ Pa_v^'W and v ∉ Pa_u^'W in G'_W then
20:            return false
21:        end if
22:    end for
23:    for each u ∈ Pa_v^'W in G'_W do
24:        if u ∉ Pa_v^W and v ∉ Pa_u^W in G_W then
25:            return false
26:        end if
27:    end for
28: end for
29: /* Check v-structures */
30: for each node v ∈ W do
31:    if CheckVStructure((v, G_W, G'_W))=false then
32:        return false
33:    end if
34: end for
35: return true
```

APPENDIX B. COMPUTING THE POSTERIORS of $s \rightsquigarrow p \rightsquigarrow t$ RELATIONS

In this appendix we provide supplementary material for chapter 5. Here we extend our algorithm to compute the exact posterior of any $s \rightsquigarrow p \rightsquigarrow t$ relation, i.e., a directed path from s to t via p, in $O(n7^{n-2})$ time and $O(4^{n-2})$ space.

B.1 Algorithm

The problem is to evaluate whether there is a directed path from s to t via p. Similarly, we would like to compute the joint probability $P(s \rightsquigarrow p \rightsquigarrow t, D)$ by

$$P(s \rightsquigarrow p \rightsquigarrow t, D) = \sum_{G: s \rightsquigarrow p \rightsquigarrow t \in G} \prod_{i \in V} B_i(Pa_i^G). \tag{B.1}$$

For any T, R, S such that $p \in T \subset R \subseteq S \subseteq V$, $s \in R - T$, let $\mathcal{G}_{s,p}(S, R, T)$ denote the set of all possible DAGs over S such that R are the set of all descendants of s (including s) and T are the set of all descendants of p (including p) in G_S. That is, $G_S \in \mathcal{G}_{s,p}(S, R, T)$ if and only if $de_{G_S}(s) = R$ and $de_{G_S}(p) = T$. We then define

$$H_{s,p}(S, R, T) \equiv \sum_{G_S \in \mathcal{G}_{s,p}(S,R,T)} \prod_{i \in S} B_i(Pa_i^{G_S}). \tag{B.2}$$

Then we have

Lemma B.1.

$$P(s \rightsquigarrow p \rightsquigarrow t, D) = \sum_{T,R: \{p,t\} \subseteq T \subset R \subseteq V, s \in R-T} H_{s,p}(V, R, T). \tag{B.3}$$

Proof. . Let $\mathcal{G}_{s \rightsquigarrow p \rightsquigarrow t} = \{G : s \rightsquigarrow p \rightsquigarrow t \in G\}$, namely the set of all possible DAGs over V that contains a $s \rightsquigarrow p \rightsquigarrow t$. Then we have $\mathcal{G}_{s \rightsquigarrow p \rightsquigarrow t} = \cup_{T,R: \{p,t\} \subseteq T \subset R \subseteq V, s \in R-T} \mathcal{G}_{s,p}(V, R, T)$.

Further, for any $T_1 \neq T_2$ or $R_1 \neq R_2$, we have $\mathcal{G}_{s,p}(V, R_1, T_1) \cap \mathcal{G}_{s,p}(V, R_2, T_2) = \emptyset$. This means $\mathcal{G}_{s,p}(V, R, T)$ for all T, R such that $p \in T \subset R \subseteq V$, $s \in R - T$ form a partition of the set $\mathcal{G}_{s \rightsquigarrow p \rightsquigarrow t}$. Thus,

$$\begin{aligned} P(s \rightsquigarrow p \rightsquigarrow t, D) &= \sum_{G \in \mathcal{G}_{s \rightsquigarrow p \rightsquigarrow t}} \prod_{i \in V} B_i(Pa_i^G) = \sum_{\substack{T,R:\{p,t\} \subseteq T \subset R \subseteq V \\ s \in R - T}} \sum_{G \in \mathcal{G}_{s,p}(V,R,T)} \prod_{i \in V} B_i(Pa_i^G) \\ &= \sum_{T,R:\{p,t\} \subseteq T \subset R \subseteq V, s \in R - T} H_{s,p}(V, R, T). \end{aligned} \tag{B.4}$$

□

If we have all $H_{s,p}(S, R, T)$ computed, it takes $\sum_{|R|=1}^{n} \left[\binom{n-3}{|R|-3} \sum_{|T|=2}^{|R|-1} \binom{|R|-3}{|T|-2} \right] = O(3^{n-3})$ time to compute Equation B.3.

Now we can show that $H_{s,p}(S, R, T)$ for all T, R, S such that $p \in T \subset R \subseteq S \subseteq V$ and $s \in R - T$ can be computed recursively. These $H_{s,p}(S, R, T)$'s can be divided into two cases: $T = \{p\}$ and $T \neq \{p\}$.

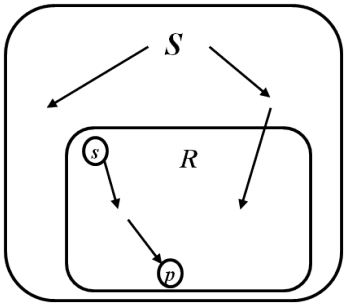


Figure B.1: Case 1: $T = \{p\}$.

Case 1: $T = \{p\}$.

In this case, p is a sink in G_S (see Figure B.1) and its parent set must include a least one node in $R - \{p\}$ to make it a descendant of s. For nodes in $S - \{p\}$, we have summation over $\mathcal{G}_s(S - \{p\}, R - \{p\})$, i.e., the set of DAGs over $S - \{p\}$ s.t. $R - \{p\}$ are the set of descendants

of s in $G_{S-\{p\}}$. Then we have

$$\begin{aligned} H_{s,p}(S,R,\{p\}) &= [\sum_{\substack{Pa_p\subseteq S-\{p\}\\ Pa_p\cap R-\{p\}\neq\emptyset}} B_p(Pa_p)][\sum_{G_{S-\{p\}}\in\mathcal{G}_s(S-\{p\},R-\{p\})}\prod_{i\in S-\{p\}} B_i(Pa_i^{G_{S-\{p\}}})] \\ &= [\sum_{Pa_p\subseteq S-\{p\}} B_p(Pa_p) - \sum_{Pa_p\subseteq S-R} B_p(Pa_p)]H_s(S-\{p\},R-\{p\}) \\ &= [A_p(S-\{p\}) - A_p(S-R)]H_s(S-\{p\},R-\{p\}) \\ &= [AA(S-\{p\},\{p\}) - AA(S-R,\{p\})]H_s(S-\{p\},R-\{p\}). \end{aligned} \tag{B.5}$$

Case 2: $T \neq \{p\}$.

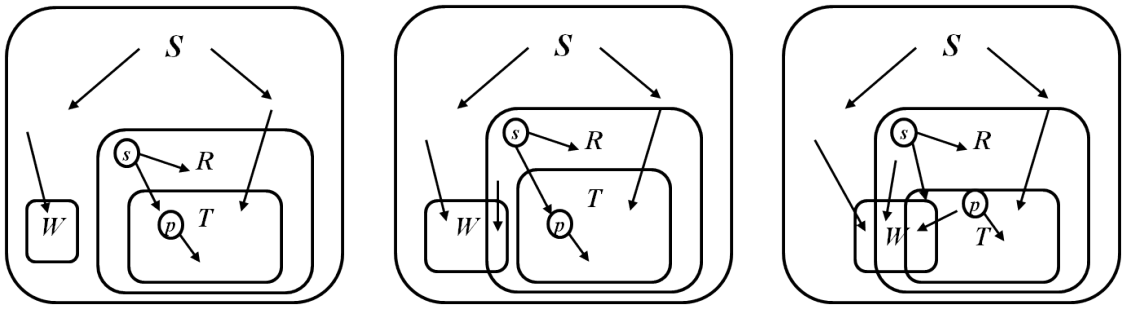
For any $W \subseteq S-\{s,p\}$, let $\mathcal{G}_{s,p}(S,R,T,W)$ denote the set of DAGs in $\mathcal{G}_{s,p}(S,R,T)$ such that all nodes in W are (must be) sinks.[1] Then we define

$$F_{s,p}(S,R,T,W) \equiv \sum_{G_S\in\mathcal{G}_{s,p}(S,R,T,W)}\prod_{i\in S} B_i(Pa_i^{G_S}). \tag{B.6}$$

Similarly, by weighted inclusion-exclusion principle,

$$\begin{aligned} H_{s,p}(S,R,T) &= \sum_{k=1}^{|S|-2}(-1)^{k+1}\sum_{W\subseteq S-\{s,p\},|W|=k}\sum_{G_S\in\mathcal{G}_{s,p}(S,R,T,W)}\prod_{i\in S} B_i(Pa_i) \\ &= \sum_{k=1}^{|S|-2}(-1)^{k+1}\sum_{W\subseteq S-\{s,p\},|W|=k} F_{s,p}(S,R,T,W). \end{aligned} \tag{B.7}$$

$F_{s,p}(S,R,T,W)$ and $H_{s,p}(S,R,T)$ can be computed recursively. There are three sub-cases (see Figure B.2).



(a) $W \cap R = \emptyset$ (b) $W \cap R \neq \emptyset$ and $W \cap T = \emptyset$ (c) $W \cap T \neq \emptyset$

Figure B.2: Three sub-cases when computing $F_{s,p}(S,R,T,W)$.

[1] Again, W may not include all the sinks in G_S. Some nodes in $S-W$ could be sinks.

Sub-case 1: $W \cap R = \emptyset$.

We can compute the summation for W and $S-W$ separately (see Figure B.2(a)). We have

$$F_{s,p}(S,R,T,W) = [\prod_{j\in W}\sum_{Pa_j\subseteq(S-R-W)} B_j(Pa_j)][\sum_{G_{S-W}\in\mathcal{G}_{s,p}(S-W,R,T)}\prod_{i\in S-W} B_i(Pa_i^{G_{S-W}})]$$

$$= \prod_{j\in W} A_j(S-R-W)H_{s,p}(S-W,R,T) = AA(S-R-W,W)H_{s,p}(S-W,R-W,T-W)$$

(because $R-W=R$ and $T-W=T$ in this case). (B.8)

Sub-case 2: $W \cap R \neq \emptyset$ **and** $W \cap T = \emptyset$.

In this case, nodes in $W-R$, $W\cap R$, and $S-W$ should be handled separately (see Figure B.2(b)). Nodes in $W-R$ can only select parents from $S-R-W$. Any node in $W\cap R$ can select parents from $S-W-T$. In addition, at least one node from $R-T-W$ must be included in its parent set to guarantee that it is a descendant of s. For nodes in $S-W$, we have summation over $\mathcal{G}_{s,p}(S-W,R-W,T)$. Then we have

$$F_{s,p}(S,R,T,W)$$

$$= [\prod_{j\in W-R}\sum_{Pa_j\subseteq(S-R-W)} B_j(Pa_j)][\prod_{j\in W\cap R}\sum_{\substack{Pa_j\subseteq(S-W-T)\\ Pa_j\cap(R-T-W)\neq\emptyset}} B_j(Pa_j)][\sum_{\substack{G_{S-W}\in\\ \mathcal{G}_{s,p}(S-W,R-W,T)}}\prod_{i\in S-W} B_i(Pa_i^{G_{S-W}})]$$

$$= \prod_{j\in W-R} A_j(S-R-W)\left\{\prod_{j\in W\cap R}[A_j(S-W-T)-A_j(S-W-R)]\right\}H_{s,p}(S-W,R-W,T)$$

$$= AA(S-W-R,W-R)\left\{\prod_{j\in W\cap R}[A_j(S-W-T)-A_j(S-W-R)]\right\}H_{s,p}(S-W,R-W,T-W)$$

(because $T-W=\emptyset$ in this case). (B.9)

Sub-case 3: $W \cap T \neq \emptyset$.

In this case, nodes in $W-R$, $W\cap(R-T)$, $W\cap T$, and $S-W$ should be handled separately (see Figure B.2(c)). Nodes in $W-R$ can only select parents from $S-R-W$. Any node in $W\cap(R-T)$ can select parents from $S-W-T$. In addition, at least one node from $R-T-W$ must be included in its parent set to guarantee that it is a descendant of s. Nodes in $W\cap T$ can select parents from $S-W$ and at least one node as its parent from $T-W$ to make it a

descendant of p. For nodes in $S-W$, we have summation over $\mathcal{G}_{s,p}(S-W, R-W, T-W)$. Then we have

$$
\begin{aligned}
&F_{s,p}(S,R,T,W) \\
&\quad = [\prod_{j\in W-R}\sum_{Pa_j\subseteq(S-R-W)} B_j(Pa_j)][\prod_{j\in W\cap(R-T)}\sum_{\substack{Pa_j\subseteq(S-W-T)\\ Pa_j\cap(R-T-W)\neq\emptyset}} B_j(Pa_j)] \\
&\quad [\prod_{j\in W\cap T}\sum_{\substack{Pa_j\subseteq(S-W)\\ Pa_j\cap(T-W)\neq\emptyset}} B_j(Pa_j)][\sum_{\substack{G_{S-W}\in\\ \mathcal{G}_{s,p}(S-W,R-W,T-W)}}\prod_{i\in S-W} B_i(Pa_i^{G_{S-W}})] \\
&= \prod_{j\in W-R} A_j(S-R-W)\left\{\prod_{j\in W\cap(R-T)}[A_j(S-W-T)-A_j(S-W-R)]\right\} \\
&\quad \left\{\prod_{j\in W\cap T}[A_j(S-W)-A_j(S-W-T)]\right\} H_{s,p}(S-W,R-W,T-W) \\
&= AA(S-W-R,W-R)\left\{\prod_{j\in W\cap(R-T)}[A_j(S-W-T)-A_j(S-W-R)]\right\} \\
&\quad \left\{\prod_{j\in W\cap T}[A_j(S-W)-A_j(S-W-T)]\right\} H_{s,p}(S-W,R-W,T-W).
\end{aligned}
\tag{B.10}
$$

For ease of exposition, for all S, R, T, W such that $\{p\}\subset T\subset R\subseteq S\subseteq V$, $s\in R-T$ and $W\subseteq S-\{s,p\}$, define function $\mathcal{A}_{s,p}(S,R,T,W)$ as follows:

If $W\cap R=\emptyset$,

$$\mathcal{A}_{s,p}(S,R,T,W)\equiv AA(S-R-W,W);$$

If $W\cap R\neq\emptyset$ and $W\cap T=\emptyset$,

$$\mathcal{A}_{s,p}(S,R,T,W)\equiv AA(S-W-R,W-R)\left\{\prod_{j\in W\cap R}[A_j(S-W-T)-A_j(S-W-R)]\right\};$$

If $W\cap T\neq\emptyset$,

$$
\begin{aligned}
\mathcal{A}_{s,p}(S,R,T,W)\equiv AA(S-W-R,W-R)&\left\{\prod_{j\in W\cap(R-T)}[A_j(S-W-T)-A_j(S-W-R)]\right\} \\
&\left\{\prod_{j\in W\cap T}[A_j(S-W)-A_j(S-W-T)]\right\}.
\end{aligned}
\tag{B.11}
$$

Now $F_{s,p}(S,R,T,W)$ can be neatly written as

$$F_{s,p}(S,R,T,W) = \mathcal{A}_{s,p}(S,R,T,W)H_{s,p}(S-W,R-W,T-W). \tag{B.12}$$

Then we have a recursive formula for computing $H_{s,p}(S,R,T)$,

$$H_{s,p}(S,R,T) = \sum_{k=1}^{|S|-2}(-1)^{k+1}\sum_{W\subseteq S-\{s,p\},|W|=k}\mathcal{A}_{s,p}(S,R,T,W)H_{s,p}(S-W,R-W,T-W). \tag{B.13}$$

And finally, we arrive the following recursive scheme for computing $H_{s,p}(S,R,T)$ for all T, R, S such that $p\in T\subset R\subseteq S\subseteq V$ and $s\in R-T$.

Theorem B.1.

For all $\{s,p\}\subseteq R\subseteq S\subseteq V$,

$$H_{s,p}(S,R,\{p\}) = [AA(S-\{p\},\{p\}) - AA(S-R,\{p\})]H_s(S-\{p\},R-\{p\})$$

For all T,R,S *such that* $\{p\}\subset T\subset R\subseteq V$ *and* $s\in R-T$,

$$H_{s,p}(S,R,T) = \sum_{k=1}^{|S|-2}(-1)^{k+1}\sum_{W\subseteq S-\{s,p\},|W|=k}\mathcal{A}_{s,p}(S,R,T,W)H_{s,p}(S-W,R-W,T-W)$$

Note that all $H_s(S-\{p\},R-\{p\})$'s can be computed recursively using Theorem 5.1.

B.2 Time and Space Complexity

Computing $H_{s,p}(S,R,\{p\})$ and $H_{s,p}(S,R,T)$ dominates the total computation time. Given all $H_s(S-\{p\},R-\{p\})$'s pre-computed, $H_{s,p}(S,R,\{p\})$ for all $\{s,p\}\subseteq R\subseteq S\subseteq V$ can be computed in $\sum_{|S|=2}^{n}\binom{n-2}{|S|-2}\sum_{|R|=2}^{|S|}\binom{|S|-2}{|R|-2} = O(3^{n-2})$ time. All other $H_{s,p}(S,R,T)$'s can be computed in

$$\begin{aligned}
&\sum_{|S|=3}^{n}\binom{n-2}{|S|-2}\left\{\sum_{|R|=3}^{|S|}\binom{|S|-2}{|R|-2}\left[\sum_{|T|=2}^{|R|-1}\binom{|R|-2}{|T|-1}|S|\cdot 2^{|S|-2}\right]\right\}\\
&=\sum_{|S|=3}^{n}\binom{n-2}{|S|-2}\left\{\sum_{|R|=3}^{|S|}\binom{|S|-2}{|R|-2}|S|\cdot 2^{|S|+|R|-4}\right\}=\sum_{|S|=3}^{n}\binom{n-2}{|S|-2}\left[|S|\cdot 2^{|S|-2}\cdot 3^{|S|-2}\right]\\
&=\sum_{|S|=3}^{n}\binom{n-2}{|S|-2}\left[|S|\cdot 6^{|S|-2}\right]<n7^{n-2}.
\end{aligned} \tag{B.14}$$

Thus, the total computation time is $O(n7^{n-2})$. The space complexity is dominated by $H_{s,p}(S,R,T)$, which is

$$\begin{aligned}
&\sum_{|S|=2}^{n}\binom{n-2}{|S|-2}\left\{\sum_{|R|=2}^{|S|}\binom{|S|-2}{|R|-2}\left[\sum_{|T|=1}^{|R|-1}\binom{|R|-2}{|T|-1}\right]\right\}\\
&=\sum_{|S|=2}^{n}\binom{n-2}{|S|-2}\left\{\sum_{|R|=2}^{|S|}\binom{|S|-2}{|R|-2}2^{|R|-2}\right\}=\sum_{|S|=3}^{n}\binom{n-2}{|S|-2}3^{|S|-2}=4^{n-2}.
\end{aligned} \tag{B.15}$$

Thus, the total space requirement is $O(4^{n-2}+3^n)$. Thus, we have the following theorem.

Theorem B.2. *The posterior probability of any $s \rightsquigarrow p \rightsquigarrow t$ relation can be computed in $O(n7^{n-2})$ time and $O(4^{n-2}+3^n)$ space.*

BIBLIOGRAPHY

Achterberg, T., Berthold, T., Koch, T., and Wolter, K. (2008). Constraint integer programming: A new approach to integrate cp and mip. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 6–20. Springer.

Acid, S. and de Campos, L. M. (2001). A hybrid methodology for learning belief networks: Benedict. *International Journal of Approximate Reasoning*, 27(3):235–262.

Alcalá, J., Fernández, A., Luengo, J., Derrac, J., García, S., Sánchez, L., and Herrera, F. Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework.

Aliferis, C. F., Tsamardinos, I., Statnikov, A. R., and Brown, L. E. (2003). Causal explorer: A causal probabilistic network learning toolkit for biomedical discovery. In *METMBS*, volume 3, pages 371–376.

Allgower, E. L. and Georg, K. (1990). *Numerical continuation methods*, volume 13. Springer-Verlag Berlin.

Ananth, G., Anshul, G., George, K., and Vipin, K. (2003). *Introduction to Parallel computing*. Boston, MA: Addison-Wesley.

Bartlett, M. and Cussens, J. (2013). Advances in Bayesian network learning using integer programming. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI-13)*, pages 182–191.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.

Björklund, A., Husfeldt, T., Kaski, P., and Koivisto, M. (2007). Fourier meets möbius: fast subset convolution. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74. ACM.

Björklund, A., Husfeldt, T., Kaski, P., and Koivisto, M. (2010). Trimmed moebius inversion and graphs of bounded degree. *Theory of Computing Systems*, 47(3):637–654.

Botelho, A., Wan, H., and Heffernan, N. (2015). The prediction of student first response using prerequisite skills. In *Learning At Scale*, pages 39–45. ACM.

Bromberg, F. and Margaritis, D. (2009). Improving the reliability of causal discovery from small data sets using argumentation. In *Journal of Machine Learning Research*, pages 301–340.

Brunskill, E. (2010). Estimating prerequisite structure from noisy data. In *Educational Data Mining 2011*.

Buntine, W. (1991). Theory refinement on Bayesian networks. In *Proceedings of the Seventh conference on Uncertainty in Artificial Intelligence*, pages 52–60. Morgan Kaufmann Publishers Inc.

Castelo, R. and Kocka, T. (2003). On inclusion-driven learning of Bayesian networks. *The Journal of Machine Learning Research*, 4:527–574.

Chen, Y., Wuillemin, P.-H., and Labat, J.-M. (2015). Discovering Prerequisite Structure of Skills through Probabilistic Association Rules Mining. In *Educational Data Mining*, pages 117–124.

Chickering, D. M. (1995). A transformational characterization of equivalent Bayesian network structures. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 87–98.

Chickering, D. M. (1996). Learning Bayesian networks is NP-complete. In *Learning from data*, pages 121–130. Springer.

Chickering, D. M. (2002a). Learning equivalence classes of Bayesian network structures. *Journal of Machine Learning Research*, 2:445–498.

Chickering, D. M. (2002b). Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554.

Cooper, G. F. and Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347.

Csisz, I. et al. (1967). Information-type measures of difference of probability distributions and indirect observations. *Studia Sci. Math. Hungar.*, 2:299–318.

Cussens, J. (2011). Bayesian network learning with cutting planes. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI-11):*, pages 153–160.

Dally, W. J. and Towles, B. P. (2004). *Principles and practices of interconnection networks*. Access Online via Elsevier.

Desmarais, M. C., Meshkinfam, P., and Gagnon, M. (2006). Learned student models with item to item knowledge structures. *User Modeling and User-Adapted Interaction*, 16(5):403–434.

Eaton, D. and Murphy, K. (2007). Bayesian structure learning using dynamic programming and MCMC. In *Proceedings of the 23th Conference on Uncertainty in Artificial Intelligence*.

Ellis, B. and Wong, W. H. (2008). Learning causal Bayesian network structures from experimental data. *Journal of the American Statistical Association*, 103(482).

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99.

Friedman, N. (1997). Learning belief networks in the presence of missing values and hidden variables. In *ICML*, volume 97, pages 125–133.

Friedman, N. and Koller, D. (2003). Being Bayesian about network structure. a Bayesian approach to structure discovery in Bayesian networks. *Machine learning*, 50(1-2):95–125.

Gillispie, S. B. and Perlman, M. D. (2001). Enumerating markov equivalence classes of acyclic digraph dels. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 171–177.

González-Brenes, J. P. (2015). Modeling Skill Acquisition Over Time with Sequence and Topic Modeling. In *International Conference on Artificial Intelligence and Statistics*, pages 296–305.

Grzegorczyk, M. and Husmeier, D. (2008). Improving the structure MCMC sampler for Bayesian networks by introducing a new edge reversal move. *Machine Learning*, 71(2-3):265–305.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Hartemink, A. J. (2001). *Principled computational methods for the validation and discovery of genetic regulatory networks*. PhD thesis, Massachusetts Institute of Technology.

Hartemink, A. J., Gifford, D. K., Jaakkola, T. S., and Young, R. A. (2002). Combining location and expression data for principled discovery of genetic regulatory network models. In *Pacific symposium on biocomputing*, volume 7, pages 437–449.

Heckerman, D. and Chickering, D. M. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. In *Machine Learning*, pages 20–197.

Heckerman, D., Geiger, D., and Chickering, D. M. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243.

Heckerman, D., Meek, C., and Cooper, G. (1997). A Bayesian approach to causal discovery. Technical report, MSR-TR-97-05, Microsoft Research.

Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classification learning.

Jaakkola, T., Sontag, D., Globerson, A., and Meila, M. (2010). Learning Bayesian network structure using lp relaxations. In *International Conference on Artificial Intelligence and Statistics*, pages 358–365.

Jiang, L., Meng, D., Zhao, Q., Shan, S., and Hauptmann, A. G. (2015). Self-paced curriculum learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Käser, T., Klingler, S., Schwing, A. G., and Gross, M. (2014). Beyond knowledge tracing: Modeling skill topologies with Bayesian networks. In *Intelligent Tutoring Systems*, pages 188–198. Springer.

Kennes, R. (1992). Computational aspects of the mobius transformation of graphs. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(2):201–223.

Koivisto, M. (2006a). Advances in exact Bayesian structure discovery in Bayesian networks. In *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence*.

Koivisto, M. (2006b). An o*(2ˆ n) algorithm for graph coloring and other partitioning problems via inclusion–exclusion. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 583–590. IEEE.

Koivisto, M. and Sood, K. (2004). Exact Bayesian structure discovery in Bayesian networks. *The Journal of Machine Learning Research*, 5:549–573.

Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Kumar, M. P., Packer, B., and Koller, D. (2010). Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems 23*.

Lazic, N., Bishop, C., and Winn, J. (2013). Structural expectation propagation (sep): Bayesian structure learning for networks with latent variables. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 379–387.

Li, J. and Wang, Z. J. (2009). Controlling the false discovery rate of the association/causality structure learned with the PC algorithm. *The Journal of Machine Learning Research*, 10:475–514.

Loh, P. K., Hsu, W.-J., and Pan, Y. (2005). The exchanged hypercube. *Parallel and Distributed Systems, IEEE Transactions on*, 16(9):866–874.

Madigan, D., Andersson, S., Perlman, M., and Volinsky, C. (1996). Bayesian model averaging and model selection for markov equivalence classes of acyclic digraphs. In *Communications in Statistics: Theory and Methods*, pages 2493–2519.

Madigan, D. and Raftery, A. E. (1994). Model selection and accounting for model uncertainty in graphical models using occam's window. *Journal of the American Statistical Association*, 89(428):1535–1546.

Madigan, D., York, J., and Allard, D. (1995). Bayesian graphical models for discrete data. *International Statistical Review/Revue Internationale de Statistique*, pages 215–232.

Malone, B., Järvisalo, M., and Myllymäki, P. (2015). Impact of learning strategies on the quality of Bayesian networks: An empirical evaluation. In *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence (UAI 2015)*.

Malone, B. and Yuan, C. (2012). A parallel, anytime, bounded error algorithm for exact Bayesian network structure learning. In *Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM-12)*.

Malone, B. and Yuan, C. (2013). Evaluating anytime algorithms for learning optimal Bayesian networks. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI-13)*.

Malone, B. M., Yuan, C., and Hansen, E. A. (2011). Memory-efficient dynamic programming for learning optimal Bayesian networks. In *AAAI*.

Margaritis, D. and Thrun, S. (2000). Bayesian network induction via local neighborhoods. In Solla, S. A., Leen, T., and Müller, K.-R., editors, *Advances in Neural Information Processing Systems 12*, pages 505–511. MIT Press.

Meek, C. (1995). Strong completeness and faithfulness in Bayesian networks. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 411–418. Morgan Kaufmann Publishers Inc.

Mislevy, R. J., Almond, R. G., Yan, D., and Steinberg, L. S. (1999). Bayes nets in educational assessment: Where the numbers come from. In *Uncertainty in artificial intelligence*, pages 437–446.

Nederlof, J. (2009). Fast polynomial-space algorithms using möbius inversion: Improving on steiner tree and related problems. In *Automata, Languages and Programming*, pages 713–725. Springer.

Niinimäki, T. and Koivisto, M. (2013). Annealed importance sampling for structure learning in Bayesian networks. In *23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*.

Niinimäki, T. M., Parviainen, P., Koivisto, M., et al. (2011). Partial order MCMC for structure discovery in Bayesian networks. In *Proceedings of the Twenty-Seventh Conference Conference on Uncertainty in Artificial Intelligence (UAI-11)*.

Nikolova, O., Zola, J., and Aluru, S. (2009). A parallel algorithm for exact Bayesian network inference. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 342–349. IEEE.

Nikolova, O., Zola, J., and Aluru, S. (2013). Parallel globally optimal structure learning of Bayesian networks. *Journal of Parallel and Distributed Computing*.

Ott, S., Imoto, S., and Miyano, S. (2004). Finding optimal models for small gene networks. In *Pacific symposium on biocomputing*, volume 9, pages 557–567.

Parviainen, P. and Koivisto, M. (2009). Exact structure discovery in Bayesian networks with less space. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI-09)*, pages 436–443.

Parviainen, P. and Koivisto, M. (2010). Bayesian structure discovery in Bayesian networks with less space. In *International Conference on Artificial Intelligence and Statistics*, pages 589–596.

Parviainen, P. and Koivisto, M. (2011). Ancestor relations in the presence of unobserved variables. In *Machine Learning and Knowledge Discovery in Databases*, pages 581–596.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Pearl, J. (2000). *Causality: models, reasoning and inference*, volume 29. Cambridge Univ Press.

Pellet, J.-P. and Elisseeff, A. (2008). Using markov blankets for causal structure learning. *The Journal of Machine Learning Research*, 9:1295–1342.

Rota, G.-C. (1964). On the foundations of combinatorial theory i. theory of möbius functions. *Probability theory and related fields*, 2(4):340–368.

Sachs, K., Perez, O., Pe'er, D., Lauffenburger, D. A., and Nolan, G. P. (2005). Causal protein-signaling networks derived from multiparameter single-cell data. *Science*, 308(5721):523–529.

Scheines, R., Silver, E., and Goldin, I. (2014). Discovering prerequisite relationships among knowledge components. In *Educational Data Mining 2014*.

Silander, T. and Myllymäki, P. (2006). A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22th Conference on Uncertainty in Artificial Intelligence*, pages 445–452.

Singh, A. P. and Moore, A. W. (2005). Finding optimal Bayesian networks by dynamic programming. Technical report, CMU-CALD-05-106, Carnegie Mellon University.

Spirtes, P., Glymour, C., and Scheines, R. (2000). *Causation, prediction, and search*, volume 81. The MIT Press.

Spirtes, P., Glymour, C., and Scheines, R. (2001). *Causation, prediction, and search*. MIT Press.

Spitkovsky, V. I., Alshawi, H., and Jurafsky, D. (2010). From baby steps to leapfrog: How "less is more" in unsupervised dependency parsing. In *NAACL*.

Tamada, Y., Imoto, S., and Miyano, S. (2011). Parallel algorithm for learning optimal Bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459.

Templin, J. and Bradshaw, L. (2014). Hierarchical diagnostic classification models: A family of models for estimating and testing attribute hierarchies. *Psychometrika*, 79(2):317–339.

Tian, J. and He, R. (2009). Computing posterior probabilities of structural features in Bayesian networks. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 538–547.

Tian, J., He, R., and Ram, L. (2010). Bayesian model averaging using the k-best Bayesian network structures. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*.

Tian, J. and Pearl, J. (2001). Causal discovery from changes. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 512–521. Morgan Kaufmann Publishers Inc.

Tsamardinos, I., Aliferis, C. F., and Statnikov, A. (2003). Time and sample efficient discovery of markov blankets and direct causal relations. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 673–678. ACM.

Tsamardinos, I., Brown, L. E., and Aliferis, C. F. (2006). The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning*, 65(1):31–78.

Tu, K. and Honavar, V. (2011). On the utility of curricula in unsupervised learning of probabilistic grammars. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1523.

VanLehn, K. (1988). Student modeling. *Foundations of intelligent tutoring systems*, 55:78.

Verma, T. and Pearl, J. (1990). Equivalence and synthesis of causal models. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence*, pages 255–270.

Vuong, A., Nixon, T., and Towle, B. (2010). A method for finding prerequisites within a curriculum. In *Educational Data Mining 2011*.

Yuan, C. and Malone, B. (2012). An improved admissible heuristic for learning optimal Bayesian networks. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI-12)*.

Yuan, C., Malone, B., and Wu, X. (2011). Learning optimal Bayesian networks using a* search. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*, pages 2186–2191.