UNIVERSITY OF CALIFORNIA

Los Angeles

Modeling, Learning and Reasoning with Structured Bayesian Networks

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Yujia Shen

2020

ABSTRACT OF THE DISSERTATION

Modeling, Learning and Reasoning with Structured Bayesian Networks

by

Yujia Shen
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Adnan Youssef Darwiche, Chair

Probabilistic graphical models, e.g. Bayesian Networks, have been traditionally introduced to model and reason with uncertainty. A graph structure is crafted to capture knowledge of conditional independence relationships among random variables, which can enhance the computational complexity of reasoning. To generate such a graph, one sometimes has to provide vast and detailed knowledge about how variables interacts, which may not be readily available. In some cases, although a graph structure can be obtained from available knowledge, it can be too dense to be useful computationally. In this dissertation, we propose a new type of probabilistic graphical models called a *Structured Bayesian network* (SBN) that requires less detailed knowledge about conditional independences. The new model can also leverage other types of knowledge, including logical constraints and conditional independencies that are not visible in the graph structure. Using SBNs, different types of knowledge act in harmony to facilitate reasoning and learning from a stochastic world. We study SBNs across the dimensions of modeling, inference and learning. We also demonstrate some of their applications in the domain of traffic modeling.

The dissertation of Yujia Shen is approved.

Qing Zhou

Songwu Lu

Kai-Wei Chang

Adnan Youssef Darwiche, Committee Chair

University of California, Los Angeles

2020

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Pursuing a doctorate has been an unforgettable journey. I am deeply blessed and honored to have the opportunity to follow this path. I first want to show my greatest appreciation to my advisor Adnan Darwiche. Thank you for leading me onto this trip and inspiring me to look at problems critically. I also want to thank Arthur Choi, who has worked with me day-to-day since the day when I started. Both Adnan and him have influenced me to be a better thinker not only about the research problems, but also about the choices of life.

I also want to thank every other member of my doctoral committee, Kai-Wei Chang, Songwu Lu, Qing Zhou for their efforts while serving on my committee. Their valuable feedback has brought me new perspectives about the significance of the problems that I tackled.

My sincere thanks also go to fellow group members, past and present: Eunice Chen, Khaled Refaat, Umut Oztok, and Doga Kisa. Their sharings of wisdoms on life have made a smooth journey of mine.

I want to thank all my other co-authors, Haiying Huang, Anchal Goyanka, and Eunice Chen, whose collaboration have made this work wonderful.

I am grateful to have made many friends from UCLA: Andy Shih, Anji Liu, Antonio Vergari, Guy Van den Broeck, Honghua Zhang, Jieyu Zhao, Kareem Ahmed, Kuan-Hao Huang, Laura Isabel Galindez Olascoaga, Liunian Li, Muhao Chen, Pasha Khosravi, Steven Holtzen, Tal Friedman, Tao Meng, Tong He, Xumeng Han, Yan Lin, Yitao Liang, YooJung Choi, and Zhe Zeng. It has been a joyful memory that we have crafted together.

Finally, I am lucky to have Chi. You are a great causality teacher, and you have also been a lively "rubber duck" that not only patiently listens to my practice talks but also provides valuable feedbacks for me to improve. Further, your dancing, cooking and, most importantly, company have made this challenging journey warm and comfortable.

VITA

2011–2014	B.S. (Computer Science Engineering) University of California, Los Angeles
2014–2015	M.S. (Computer Science) University of California, Los Angeles

PUBLICATIONS

Yujia Shen, Arthur Choi and Adnan Darwiche. A New Perspective on Learning Context-Specific Independence, *In International Conference on Probabilistic Graphical Models,* 2020.

Yujia Shen, Haiying Huang, Arthur Choi and Adnan Darwiche. Conditional Independence in Testing Bayesian Networks, *In Proceedings of 36th ICML,* 2019.

Yujia Shen, Arthur Choi and Adnan Darwiche. Structured Bayesian Networks: From Inference to Learning with Routes, *In Proceedings of the 33rd AAAI,* 2019.

Yujia Shen, Arthur Choi and Adnan Darwiche. Conditional PSDDs: Modeling and Learning with Modular Knowledge, *In Proceedings of the 32nd AAAI,* 2018.

Yujia Shen, Arthur Choi and Adnan Darwiche. A Tractable Probabilistic Model for Subset Selection, *In Proceedings of the 33rd UAI,* 2017.

Arthur Choi, **Yujia Shen** and Adnan Darwiche. Tractability in Structured Probability Spaces, *In Advances in Neural Information Processing Systems 30 (NIPS),* 2017.

Yujia Shen, Arthur Choi and Adnan Darwiche. Tractable Operations for Arithmetic Circuits of Probabilistic Models, *in Advances in Neural Information Processing Systems 29 (NIPS),* 2016.

Eunice Yuh-Jie Chen, **Yujia Shen**, Arthur Choi and Adnan Darwiche. Learning Bayesian Networks with Ancestral Constraints, *In Advances in Neural Information Processing Systems 29 (NIPS),* 2016.

CHAPTER 1

Introduction

Researching a problem usually involves discovering its structure. For example, Newton's Laws of Motion are essentially discovering the structure among three factors: force, acceleration, and velocity. Using the structure, we can understand why the apple falls to the ground, and we can also predict the location of our earth in the next decade. There are different types of structure one can explore. For each of the structure, a model is usually proposed to especially exploit it for learning and reasoning purpose.

Bayesian networks (BNs) have been proposed to capture problem structure in the form of conditional independence [Dar09, KF09, Bar12]. In particular, a DAG is used to model the conditional independence relationships in the problem. Consider a simple pandemic model containing three variables: C (COVID-19), I (Influenza) and F (fever). The DAG in Figure 1.1 demonstrates the conditional independence relationships among the three random variables. Influenza and COVID-19 are two types of viral diseases, which can independently infect a person. Further, we have also known that COVID-19 and Influenza can cause similar symptoms, i.e. fever. The dependency between the symptom of fever and the infections of either virus is also probabilistic, as the manifestation of the fever also depends on many unobserved factors, including the physical wellness of the individual. From the conditional independence, the joint probability distribution can be decomposed into three smaller pieces:

$$Pr(C, I, F) = Pr(C)Pr(I)Pr(F \mid C, I).$$

BNs can elegantly model the simple epidemic model by exploiting the conditional independence structure, but some other problems may not exhibit any of this structure. If BNs were still used to model those problems, the DAG would be fully connected, without exploit-

$$\begin{tikzcd}
\text{COVID-19} \arrow[dr] & & \text{influenza} \arrow[dl] \\
& \text{fever} &
\end{tikzcd}$$

Figure 1.1: A Bayesian Network that models the simplified COVID-19 diagnosis.

ing any problem structure. As a result, the BN needs to specify a parameter for each joint assignment of the variables, and there are an exponential number of these joint assignments to the number of variables that are modeled [Dar09, Pea89]. Let's consider a store in a fish market. Inside the store, a fishmonger sells fish that it has caught from the ocean. As there is limited room to stock the fish, the fishmonger has to select a fixed amount of fish for sale. As the fishmonger catches different types of fish every day, the selected fish also change from time to time. If the stochasticity of this problem is modeled using a BN, the DAG, that describes the joint distribution over the selected fish types, must be fully connected. In particular, each node in the DAG indicates whether a particular fish is selected for sale, and the selection of any fish depends on the availability of the space, which ultimately depends on the number of all the other fish that are selected. For example, Figure 1.2 shows a DAG that models the selection problem with 20 fish. The DAG is so densely connected that no problem structure is revealed. Further, the BN with this DAG needs $2^{20} \approx 10^6$ parameters to model the uncertainties of the selection. This ultimately creates computational challenges for learning and reasoning.

The fish selection problem does not exhibit any structure of conditional independence, and BNs are not suited to model these problems. This motivates us to explore a different type of structure, called *context-specific independence* [BFG96], which corresponds to local structure in conditional probabilities. Instead of specifying a distinct conditional probability for each child variable given each parent configuration, many parent configurations share the same child probability. For example, in the fishmonger example, the selection of any fish may only depend on *the number* of the other fish that are selected. To specify such a conditional probability, one only needs a *linear* number of parameters; a single parameter is needed for each cardinality of the selection among the parent fish. As a pure BN needs an exponential number of parameters, one for each assignment to the parent and child variables, it is a significant improvement to exploit the local structure in the conditional probability distribution. Probabilistic Sentential Decision Diagrams (PSDDs) have been introduced to model these context-specific independences even when the DAG can be fully connected [KVC14]. PSDDs are effective in modeling many combinatorial objects, e.g. rankings [CVD15] and game trajectories [CTD16], where conditional independences cannot be exploited at the structural level.

Although PSDDs can model complex objects by exploiting context-specific independences, they cannot model the topological conditional independences, that BNs can model. In the fish market example, one might want to model a joint distribution over the selections of both the fishmonger and its customers. The selection of a customer is corrected with the selection of the fishmonger. For example, if a popular kind of fish is offered in-store by the fishmonger, each customer is more likely to select it. Furthermore, we have some topological conditional independences among the selections of the customers; each customer usually makes its selection independently. In this problem, we want to exploit both structures: context-specific independence to model each of the selections, and topological independence to model the relationship between the choices of customers and the fishmonger. However, PSDDs cannot directly exploit both of these structures.

To address this problem formally, we are proposing a new framework called structured Bayesian networks (SBNs). In a nutshell, it uses a DAG to capture the probabilistic dependency among different complex objects, where each can be represented using multiple variables. For example, Figure 1.3 shows a DAG that describes the dependency among different selections in the fish market example. The root node contains variables that describe the fish that are selected by the fishmonger, and each leaf node contains variables that describe the fish that are selected by each customer. The DAG describes the topological independence; given fishmonger's selection, customers' selections are independent of each other. At the same time, the DAG keeps silent about dependency among variables in the same node, which are used to represent a selection. It offers opportunities to model each selection with context-specific independence. SBNs inherit the benefits from both of its ancestors: BNs and PSDDs. Similar to BNs, SBNs follows a global DAG structure to decompose a joint probability into small conditional distributions. Further, a PSDD can be used to model each conditional distribution with knowledge of context-specific independence, which traditional BNs do not exploit. We will also demonstrate the utility of this marriage between BNs and PSDDs in the application of tractable route predictions, which cannot be modeled only using either PSDDs or BNs.

Next, we provide an overview of the remaining chapters of this dissertation.

In Chapter 2, we review Probabilistic Sentential Decision Diagrams (PSDDs). We first show their syntax and semantics, and we end the chapter with some of the probabilistic queries that PSDDs can solve efficiently.

In Chapter 3, we consider the problem of using PSDDs to model combinatorial objects. The combinatorial object is an n-choose-k selection, which admits the selections of a fishmonger in the fish market as a special case.

In Chapter 4, we formally introduce structured Bayesian networks, which is a new model that we are proposing. Each node in a structured Bayesian network contains a set of variables, as in Figure 1.3. These variables can be used to describe a combinatorial object, e.g. selection. Besides, we introduce a new conditional model, called *conditional PSDDs,* to model the dependencies among different combinatorial objects.

In Chapter 5, we study the problem of inference with SBNs. We demonstrate an exact inference algorithm for SBNs by compiling a given SBN to a PSDD. The PSDD describes the same joint probability as the SBN, and it can compute multiple probabilistic queries efficiently. The inference algorithm allows us to ask interesting questions about the joint distribution that is represented by an SBN. For example, one can compute the most likely selection of the fishmonger given the selection of a customer.

In Chapter 6, we demonstrate an application of modeling a probability distribution over routes of a map. Similar to selections, routes are also combinatorial objects. We demonstrate a decomposition of the route distribution into smaller modules by hierarchically partitioning the map. We further identify a tractable subclass of this model by using a special partitioning scheme. This tractable subclass allows one to answer queries, including the most likely route completion given a partial trip, efficiently even with a map that covers the entire road system of a city.

In Chapter 7, we study the problem of learning the structure of a conditional PSDD from data. We propose a learning algorithm that extracts the structure of context-specific independence from data. Subsequently, this structure is represented using a conditional PSDD.

We finally conclude with a summary of the thesis in Chapter 8.

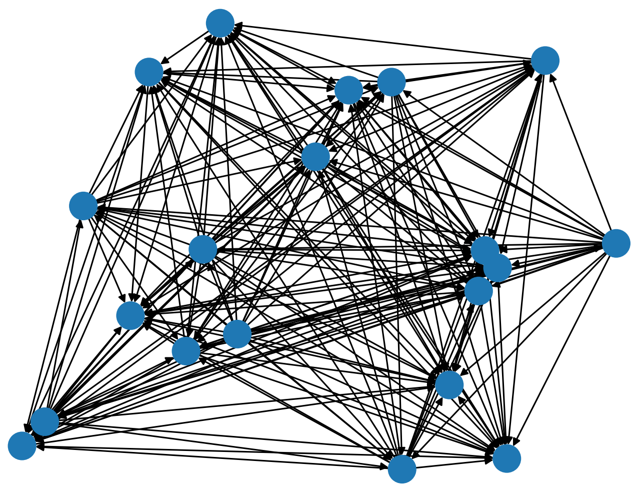


Figure 1.2: A BN representing the selection of 20 fish.

$$\begin{tikzcd}
& B_{\text{bluefin}}, B_{\text{shrimp}}, \cdots, B_{\text{albacore}} \arrow[dl] \arrow[dr] & \\
C_{\text{bluefin}}, C_{\text{shrimp}}, \cdots, C_{\text{albacore}} & & D_{\text{bluefin}}, D_{\text{shrimp}}, \cdots, D_{\text{albacore}}
\end{tikzcd}$$

Figure 1.3: A DAG that describes the graphical assumptions between the fishmonger's selection and the selections of 2 customers, Cindy and David, in the fish market. The root node contains variables that describe the fishmonger's selection, and each leaf node contains variables that describe each customer's selection.

CHAPTER 2

Probabilistic Sentential Decision Diagrams

In this chapter, we are going to review a representation of a probabilistic distribution, called Probabilistic Sentential Decision Diagrams.

2.1 Background

PSDDs have been motivated by the need to bridge probability and logic [KVC14]. Consider a joint probability as shown in Figure 2.1a. The joint probability assigns zero probability to some of the entries, e.g. $A=0$, $B=1$, $C=0$. The zero probability indicates that these configurations are impossible to happen. Traditionally, the "impossibility" is represented using a propositional logic formula. The formula only evaluates true on the possible configurations. For example, a formula that represents the logical constraint in the joint distribution is as following:

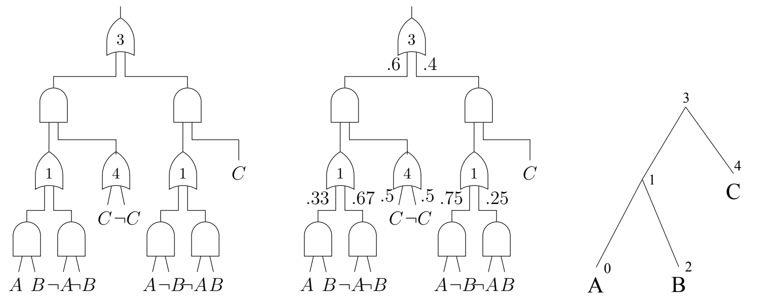
$$(A \vee \neg B \vee C) \wedge (\neg A \vee B \vee C).$$

Although a logical formula distinguishes between the "possible" and the "impossible", it does not signifies the differences in likelihood among the "possible".

PSDDs are introduced to add uncertainties among the possible configurations that are described by a logical formula. We will formally introduce its syntax and semantic in the next section. Later we will show that PSDDs are a tractable representation, where many probabilistic queries can be computed in time that scales linearly in the size of the PSDD.

A	B	C	Pr
0	0	0	**0.2**
0	0	1	**0.2**
0	1	0	0.0
0	1	1	**0.1**
1	0	0	0.0
1	0	1	**0.3**
1	1	0	**0.1**
1	1	1	**0.1**

(a) Distribution



(b) SDD (c) PSDD (d) Vtree

Figure 2.1: A probability distribution and its SDD/PSDD representation. The numbers annotating or-gates in (b) & (c) correspond to vtree node IDs in (d). Moreover, while the circuit appears to be a tree, the input variables are shared and hence the circuit is not a tree.

2.2 Syntax and Semantics

In this dissertation, we use uppercase letter, e.g. X, to represent a variable and lowercase letter, e.g. x, to represent a value of the corresponding variable. Bold case letters, i.e. $\mathbf{X}$ and $\mathbf{x}$, represents a set of variables and values respectively. Consider the distribution $Pr(\mathbf{X})$ in Figure 2.1a for an example. The first step in constructing a PSDD for this distribution is to construct a special Boolean circuit that captures its zero entries; see Figure 2.1b. The Boolean circuit captures zero entries in the following sense. For each instantiation $\mathbf{x}$, the circuit evaluates to 0 at instantiation $\mathbf{x}$ iff $Pr(\mathbf{x}) = 0$. The second and final step of constructing a PSDD amounts to parameterizing this Boolean circuit (e.g., by learning from data), which amounts to including a local distribution on the inputs of each or-gate; see Figure 2.1c.

The Boolean circuit underlying a PSDD is known as a Sentential Decision Diagram (SDD) [Dar11]. Understanding SDD circuits is key to understanding PSDDs so we review

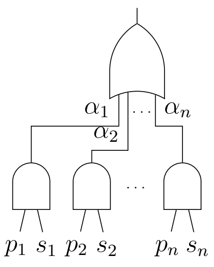


Figure 2.2: An SDD fragment.

these circuits next.

First, an SDD circuit is constructed from the fragment shown in Figure 2.2, where the or-gate can have an arbitrary number of inputs, and the and-gates have precisely two inputs each. Here, each p_i is called a **prime** and each s_i is called a **sub.** For example, the SDD circuit in Figure 2.1b is made up of three of these fragments and terminal SDDs[1].

Next, each SDD circuit conforms to a tree of variables (called a *vtree*), which is just a binary tree whose leaves are the circuit variables; see Figure 2.1d. The conformity is roughly as follows. For each SDD fragment with primes p_i and subs s_i, there must exist a vtree node v where the variables of SDD p_i are those of the left child of v and the variables of SDD s_i are those of the right child of v. The formal definition of conformity is shown in Definition 1.

Definition 1 (Conformity)**.** *An SDD circuit n is said to conform to a vtree v iff*

- *v is a leaf with variable X, and α is a terminal SDD over variable X.*
- *v is internal, and n is an SDD fragment (see Figure 2.2), where the primes $p_1, ..., p_n$ are SDDs that conform to the left vtree v^l, and the subs $s_1, ..., s_n$ are SDDs that conform to the right vtree v^r.*

For the SDD in Figure 2.1b, each or-gate has been labeled with the ID of the vtree node it conforms to. For example, the top fragment conforms to the vtree root (ID=3), with its primes having variables $\{A, B\}$ and its subs having variables $\{C\}$. SDDs that conform to

[1]A terminal SDD is either a variable (X), its negation ($\neg X$), false ($\bot$), or true (an or-gate with inputs X and $\neg X$)

a vtree were called *normalized* in [Dar11], which also defined *compressed* SDDs. These are SDDs in which the subs of a fragment are distinct.

The final key property of an SDD circuit is this. When the circuit is evaluated under *any* input, precisely one prime p_i of each fragment will be 1. Hence, the fragment output will simply be the value of the corresponding sub s_i.[2]

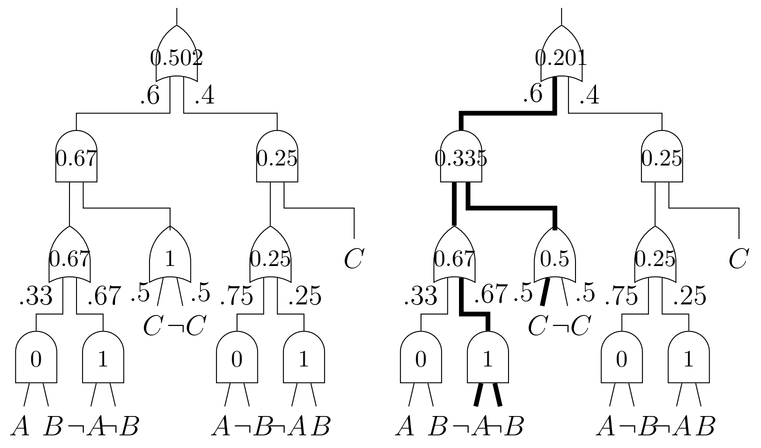
A PSDD can now be obtained by annotating a distribution $\alpha_1, \ldots, \alpha_n$ on the inputs of each or-gate, where $\sum_i \alpha_i = 1$; see again Figure 2.2. The distribution specified by a PSDD is as follows. Let $\mathbf{x}$ be an instantiation of the PSDD variables and suppose that we evaluate the underlying SDD circuit at input $\mathbf{x}$. If the SDD evaluates to 0, then $Pr(\mathbf{x}) = 0$. Otherwise, $Pr(\mathbf{x})$ is the product of all parameters encountered by starting at the output or-gate, and then descending down to every gate and circuit input that evaluates to 1. This PSDD distribution must be normalized as long as the local distributions on or-gates are normalized [KVC14].

2.3 PSDD Queries

In this section we are demonstrating procedures that compute the results of two different probabilistic queries, a marginal query and a most probable explanation query, on PSDDs. For each query, we convert a PSDD to a computation graph. After carefully setting the values of the leaf nodes, which are the inputs to the computation graph, the root node of the computation graph, which is also the output node, will compute the query result.

Each query computation runs in time that scales linearly in the size of the corresponding PSDD. As a result, PSDDs are commonly used as compilation targets of graphical models, as we will show in Chapter 5. Given a graphical model, we can first obtain a PSDD representing the same joint distribution. This process is called compilation. After the PSDD is obtained, probabilistic queries, which are NP-hard to compute for graphical models, can be answered

[2]This implies that an or-gate will never have more than one 1-input. Also note that an SDD circuit may produce a 1-output for every possible input. These circuits arise when representing strictly positive distributions (with no zero entries).



(a) Probability Evaluation. (b) MPE Evaluation.

Figure 2.3: Figure 2.3a shows the value of each gate when computing $Pr(A=0)$, whose value is output by the root gate. Figure 2.3b shows the value of each gate when computing the MPE under observation $A=0$. The highlighted wires are visited during the reverse traversal that obtains the MPE configuration.

efficiently using the compiled PSDD.

2.3.1 Marginal Query

A joint probability distribution assigns a probability to each complete variable configuration, and a marginal query computes the probability of a partial variable configuration. A partial configuration is an assignment of a subset of variables. For example both $A=0$ and $A=0$, $B=0$ are partial configurations, and $A=0$, $B=0$, $C=0$ is an example of a complete configuration. The probability of a partial configuration equals to the sum of the probabilities of complete configurations that are compatible with it. If the joint distribution is over variables $\mathbf{X}$ and $\mathbf{Y}$, the probability of partial configuration $\mathbf{x}$ is defined as $Pr(\mathbf{x}) = \sum_{\mathbf{y}} Pr(\mathbf{x}, \mathbf{y})$.

To evaluate the probability of a partial configuration in a PSDD, we view each and-gate as a product node. Each or-gate is viewed as a weighted sum node, where the weight is the parameter annotating each input. For example, the or-gate in Figure 2.2 evaluates to $\sum_i \alpha_i \cdot p_i \cdot s_i$. This generates a computation graph from a PSDD circuit, and it computes the marginal query. Next we will set the value of leaf nodes, which are literals.

If a literal is consistent with the partial configuration, we set the literal to 1. Otherwise, the literal is set to 0. For example, given a partial configuration $A = 0$, every literals except literal A is consistent with it; hence, each consistent literal is assigned with value 1, and literal A is assigned with value 0. Another partial configuration $A = 0, B = 0$ sets literals $\neg A$, $\neg B$, C, $\neg C$ to be 1, and the remaining literals are set to 0.

After invoking the corresponding computation graph under the inputs that are properly configured, the constructed computation graph evaluates the probability of the partial configuration. For example, Figure 2.3a shows the values that are computed by each gate when evaluating the marginal query $Pr(A = 0)$. The value of the output gate, 0.502, is the result of the query.

2.3.2 Most Probable Explanation

The Most Probable Explanation (MPE) is a query that computes the most likely complete configuration that is compatible with an observation, and the observation is represented as a partial configuration. In this section, we show the computation graph for evaluating the probability of this most likely complete configuration.

To evaluate the probability of the MPE, we view each and-gate as a product node. Each or-gate is viewed as a weighted max node, where the weight is the parameter annotating each input. For example, the or-gate in Figure 2.2 evaluates to $\max_i \alpha_i \cdot p_i \cdot s_i$.

According to the observation, which is a partial configuration, the values of leaf literals are assigned in the same way as we have described in Section 2.3.1. The leaf literals are set to 1 if they are consistent with the observation and are set to 0 otherwise. At last, the computation graph will evaluate the probability of the most probable explanation that is consistent with the observation. Figure 2.3b shows the value of each gate when one evaluates the MPE query under the observation $A = 0$. The probability of the MPE is 0.201, which is the value of the output gate.

After having evaluated the probability of the MPE, we can obtain the MPE configuration by reversely traversing the computation graph. For each or-gate, one follows the branch which results in the weighted max. For each and-gate, one follows both branches. The traversal stops when it hits leaf literals. The concatenation of these literals is the MPE configuration. In Figure 2.3b, the bold wire represents the traversal, and the MPE configuration is $A=0$, $B=0$, $C=1$.

CHAPTER 3

Modeling Combinatorial Objects using PSDDs

In this chapter, we present an example of modeling combinatorial objects using PSDDs. The combinatorial object that we are interested is an n-choose-k selection. We first explain the model without using any language of PSDDs and lastly unveil their connection. This modeling example demonstrates the interpretability of PSDDs and preludes our usage of it in a more symbolic setting. The results that are discussed in this chapter appeared in [SCD17].

3.1 Introduction

We consider in this chapter the problem of selecting k items from among a set of n alternatives. This *subset selection* problem appears in a number of domains, including resource allocation (what resources shall I allocate?), preference learning (which items do I prefer?), human computation (which labelers should I recruit for my task?), and sports (which players shall play the next game?). In these subset selection tasks, a dataset consists of examples in which a fixed number (k) of variables is set to true from among a total of n variables representing choices. Given such a dataset, the goal is to learn a generative probabilistic model that can accurately represent the underlying processes that govern these selections.

Commonly used representations such as Bayesian and Markov networks are not well-suited for learning from this type of data. In general, the underlying cardinality constraints would lead to fully-connected (and hence intractable) structures—hence more specialized representations are needed to model such subset selection tasks. Recently, a new type of probabilistic model was proposed, called the n-choose-k model [STA12], that can take into account datasets whose examples have a known cardinality. The proposal includes a view on the popular logistic regression model as a mixture of n-choose-k models (with a component for each k from 0 to n). Both inference and learning are tractable in the n-choose-k model.

In this chapter, we propose a more expressive model for subset selection processes, called the *recursive n-choose-k model,* which we derived from a more general tractable representation called the Probabilistic Sentential Decision Diagram (PSDD) [KVC14]. Our proposed model is *tractable* as it can accommodate a variety of probabilistic queries in polynomial time. It is also *highly interpretable* as its parameters have precise meanings and its underlying structure explicates a generative process for the data. This is in contrast to similar tractable representations such as Arithmetic Circuits (ACs) [LD08, LR13, BDC15] and their Sum-Product Networks (SPNs) variant [PD11, GD12, DV15]. We propose a simplified closed-form parameter estimation algorithm for our recursive n-choose-k model, as well as a simple but effective structure learning algorithm. Empirically, we show how our recursive n-choose-k model is more expressive and can provide a better fit for datasets with known cardinalities than previously proposed models for this task.

This chapter is organized as follows. In Section 3.2, we review the subset selection problem and a previously proposed model. In Section 3.3, we introduce our recursive n-choose-k model. In Section 3.4, we present the corresponding parameter and structure learning algorithms. In Section 3.5, we evaluate our model empirically, and present some case studies. In Section 3.6, we show how the proposed n-choose-k model corresponds to a PSDD. Section 3.8 closes with some concluding remarks. Proofs are provided in the Appendix 3.A.

3.2 N-Choose-K Models

As a running example, consider the subset selection problem that computer science (CS) students regularly face: selecting k out of n courses to take in a quarter. For simplicity, say that students select three out of the following six courses:

learning (`ML`)	computability (`CP`)	linear algebra (`LA`)
logic (`LG`)	complexity (`CX`)	calculus (`CL`)

By column, we have two AI classes, two CS theory classes, and two math classes.

Let us now consider a dataset over students and the courses that they select. We have six variables $(\texttt{ML}, \texttt{LG}, \texttt{CP}, \texttt{CX}, \texttt{LA}, \texttt{CL})$ representing the classes that a student can select, where $\texttt{ML}=1$ means the student selected machine learning, whereas $\texttt{ML}=0$ means they did not. Our dataset consists of examples such as:

$$\texttt{ML}=0, \texttt{LG}=1, \texttt{CP}=1, \texttt{CX}=1, \texttt{LA}=0, \texttt{CL}=0$$
$$\texttt{ML}=1, \texttt{LG}=1, \texttt{CP}=0, \texttt{CX}=0, \texttt{LA}=1, \texttt{CL}=0$$
$$\texttt{ML}=1, \texttt{LG}=0, \texttt{CP}=0, \texttt{CX}=1, \texttt{LA}=0, \texttt{CL}=1.$$

Since students must select three out of six classes, each example in the dataset has exactly three positive entries, and thus three negative entries as well. We refer to such a dataset as an *n-choose-k* dataset: each example has exactly k out of n variables appearing positively, where k is called the example *cardinality.* A CS department with student enrollment data of this type may want to analyze this data and reason about the courses that students choose to take.

In this chapter, we assume that the cardinality k is known a priori. For example, in preference learning, we may be provided with data where users have selected their top-10 favorite movies, in which case exactly 10 variables appear positively, and the rest appear negatively.

[STA12] proposed a probabilistic model, called the *n-choose-k* model, which assumes a prior over k. A simpler form is obtained when k is fixed, leading to the following distribution over a set of n variables $\mathbf{X}$:

$$Pr_k(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{Z_k(\boldsymbol{\theta})} \prod_{X \in \mathbf{X}} \exp\{\theta_X \cdot \lambda_X\} \tag{3.1}$$

for instantiations $\mathbf{x}$ with cardinality k; $Pr(\mathbf{x}) = 0$ otherwise. First, $\boldsymbol{\theta} = (\ldots, \theta_X, \ldots)$ is a vector of n parameters, one parameter θ_X for each variable $X \in \mathbf{X}$. Next, we have a vector $(\ldots, \lambda_X, \ldots)$ of n indicators, one indicator λ_X for each variable $X \in \mathbf{X}$, where λ_X is 1 if $\mathbf{x}$ sets X positively and 0 otherwise. Finally, $Z_k(\boldsymbol{\theta}) = \sum_{\mathbf{x}:\mathsf{Card}(\mathbf{x})=k} \prod_{X \in \mathbf{X}} \exp\{\theta_X \cdot \lambda_X\}$ is the normalizing constant, where $\mathsf{Card}(\mathbf{x})$ is the cardinality of $\mathbf{x}$. When we take a mixture of these models, for k from 0 to n, we obtain the class conditional distribution of the logistic regression model [STA12]:

$$Pr(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{X \in \mathbf{X}} \exp\{\theta_X \cdot \lambda_X\}$$

for all instantiations $\mathbf{x}$ (of any cardinality), where $Z(\boldsymbol{\theta}) = \prod_{X \in \mathbf{X}} (1 + e^{\theta_X})$. Hence, we refer to the model of Equation 3.1 as the *logistic n-choose-k* model.

Example 1. In our course selection problem, every course X has a parameter θ_X, where a larger parameter value corresponds to a higher course popularity. For example, the parameters $(\theta_{\texttt{ML}}, \theta_{\texttt{LG}}, \theta_{\texttt{CP}}, \theta_{\texttt{CX}}, \theta_{\texttt{LA}}, \theta_{\texttt{CL}}) = (3, 2, 1, -1, -2, -3)$ suggest that `ML` is the most popular course. The probability of a student selecting machine learning (`ML`), logic (`LG`) and computability (`CP`) is then

$$\begin{aligned} &Pr(\texttt{ML}=1, \texttt{LG}=1, \texttt{CP}=1, \texttt{CX}=0, \texttt{LA}=0, \texttt{CL}=0) \\ &= \tfrac{1}{Z} \exp\{\theta_{\texttt{ML}} + \theta_{\texttt{LG}} + \theta_{\texttt{CP}} + 0 + 0 + 0\} = \tfrac{1}{Z} \exp\{6\} \end{aligned}$$

where $Z \approx 529.06$ is a normalization constant. $\square$

Next, we propose a more refined model that views course selection as a recursive process.

3.3 Recursive N-Choose-K Models

Consider the following recursive process of course selection that a student may follow, which traces the tree structure of Figure 3.1. First, at the root $\mathsf{courses}$, a student decides *how many* cs classes to take, compared to the number of math classes to take, out of a total of 3 classes. Say the student decides to take 2 cs classes and 1 math class. Following the left branch, the student decides *how many* classes to take, now between ai and theory. Suppose they take one ai class, and hence one theory class. The student then recursively decides between learning (`ML`) and logic (`LG`), and independently, between computability (`CP`) and complexity

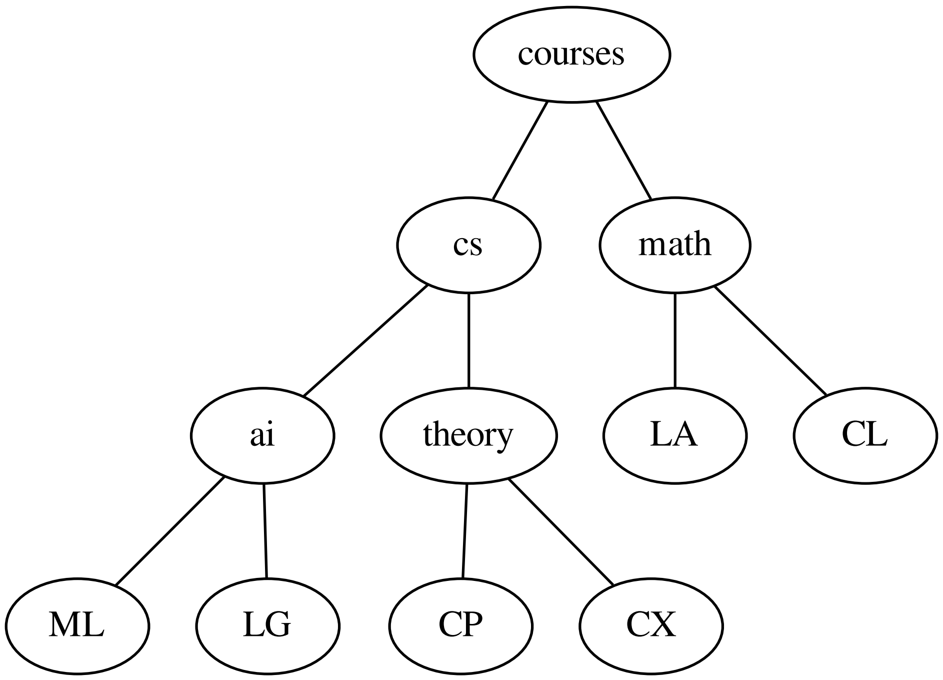


Figure 3.1: A tree hierarchy of courses.

(`CX`). We backtrack and decide (independently of the prior choices) between linear algebra (`LA`) and calculus (`CL`).

Algorithm 1 describes a probabilistic generative process for subset selection, based on a tree structure similar to the one in Figure 3.1. This structure is called a variable tree, or *vtree,* and corresponds to a full, binary tree with each leaf node v labeled with a distinct variable $X \in \mathbf{X}$. For a vtree node v, we will use $\mathbf{X}_v$ to denote the set of variables appearing at or below node v. We will also use v_1 and v_2 to represent the left and right children, respectively, of an internal vtree node v.

A call to $\mathsf{Sample}(v, k)$ of Algorithm 1 randomly selects k variables from $\mathbf{X}_v$. If v is an internal node, we first sample a cardinality k_1 of variables to select from the left child of v (which implies that we select $k_2 = k - k_1$ variables from the right child of v). If v is a leaf node, then k is either 0 (we do not select a variable), or 1 (we select the variable X at node v).

This generative process describes our new probabilistic model of subset selection, which we call the *recursive n-choose-k model.* We formalize this model next.

Algorithm 1 Sample(v, k)

```
input: Node $v$ in a vtree and a number $k$, $0 \leq k \leq |\mathbf{X}_v|$
output: A selection of $k$ variables from $\mathbf{X}_v$
main:
1: if $v$ is a leaf node labeled with variable $X$ then
2:     return $\{X=0\}$ if $k=0$ else return $\{X=1\}$
3: else
4:     $v_1, v_2 \leftarrow$ children of node $v$
5:     $\theta_{v,k} \leftarrow$ distribution over $(k_1, k_2)$ s.t. $k_1 + k_2 = k$
6:     $(k_1, k_2) \leftarrow$ a cardinality pair drawn from $\theta_{v,k}$
7:     return Sample$(v_1, k_1) \cup$ Sample$(v_2, k_2)$
```

3.3.1 Formal Definition

From here on, we assume that $n \geq 1$ and $0 \leq k \leq n$.

To define our recursive n-choose-k model, we first need to define the notion of a *choice distribution*, for deciding how many elements to choose. Such a distribution is defined for three integers n_1, n_2 and k where $n_1, n_2 \geq 1$ and $0 \leq k \leq n_1 + n_2$. The domain of this distribution is the set of pairs (k_1, k_2) such that $k_1 \leq n_1$, $k_2 \leq n_2$ and $k_1 + k_2 = k$. The intuition behind a choice distribution is this. We need to select k items from $n_1 + n_2$ items. Each pair (k_1, k_2) corresponds to a choice of k_1 items from the n_1 items and a choice of k_2 items from the n_2 items. The n_1 and n_2 items will be the variables appearing in the left and right subtrees of a vtree node v; that is, $n_1 = |\mathbf{X}_{v_1}|$ and $n_2 = |\mathbf{X}_{v_2}|$. Hence, we will denote the parameters of a choice distribution by $\theta_{v,k}(k_1, k_2)$, which represents the probability that we will select k_1 items from the left subtree of v and k_2 items from the right subtree of v. This implies that $k \leq |\mathbf{X}_v|$.

Example 2. Consider the following choice distributions from our course selection problem (we are only showing choices with non-zero probability).

v	k	k_1, k_2	$\theta_{v,k}$
courses	3	$1, 2$	0.1
		$2, 1$	0.3
		$3, 0$	0.6
math	2	$1, 1$	1.0
math	1	$1, 0$	0.3
		$0, 1$	0.7
math	0	$0, 0$	1.0

v	k	k_1, k_2	$\theta_{v,k}$
cs	2	$0, 2$	0.3
		$1, 1$	0.6
		$2, 0$	0.1
cs	1	$1, 0$	0.4
		$0, 1$	0.6
ai	1	$1, 0$	0.4
		$0, 1$	0.6

For $v =$ courses and $k = 3$, the choice distribution $\theta_{v,k}$ is used to select 3 courses from cs and math. For example, we select 1 course from cs and 2 from math with probability $\theta_{v,k}(1, 2) = 0.1$. □

We are now ready to define our subset selection model. A recursive n-choose-k model over n binary variables $\mathbf{X}$ has two components: (1) *structure:* a vtree where each leaf node is labeled with a distinct variable from $\mathbf{X}$, and (2) *parameters:* for each internal vtree node v with m leaves, a choice distribution $\theta_{v,i}$ for each $i = \max(0, k - (n - m)), \dots, \min(k, m)$. In a recursive n-choose-k model, we will never choose more than k items at any vtree node v, hence we need choice distributions for at most $\min(k, m)$ items at node v. Moreover, since we can choose at most $n - m$ items from outside node v, we must choose at least $k - (n - m)$ items at node v. Hence, we do not need choice distributions for fewer items than $\max(0, k - (n - m))$.

The distribution induced by a recursive n-choose-k model is defined inductively, over instantiations $\mathbf{x}$ whose cardinalities are k. Note that for the inductive cases, we refer to cardinalities by i rather than by k.

For the base case of a leaf vtree node v labeled with variable X, we have $Pr_{v,i}(X = \mathsf{true}) = 1$ if $i = 1$ and 0 if $i = 0$. For the inductive case of an internal leaf node v:

$$Pr_{v,i}(\mathbf{x}_v) = Pr_{v_1,i_1}(\mathbf{x}_{v_1}) \cdot Pr_{v_2,i_2}(\mathbf{x}_{v_2}) \cdot \theta_{v,i}(i_1, i_2).$$

Here, $\mathbf{x}_{v_1}$ and $\mathbf{x}_{v_2}$ are the subsets of instantiation $\mathbf{x}_v$ pertaining to variables $\mathbf{X}_{v_1}$ and $\mathbf{X}_{v_2}$, respectively. Moreover, i_1 and i_2 are the cardinalities of instantiations $\mathbf{x}_{v_1}$ and $\mathbf{x}_{v_2}$, respec-

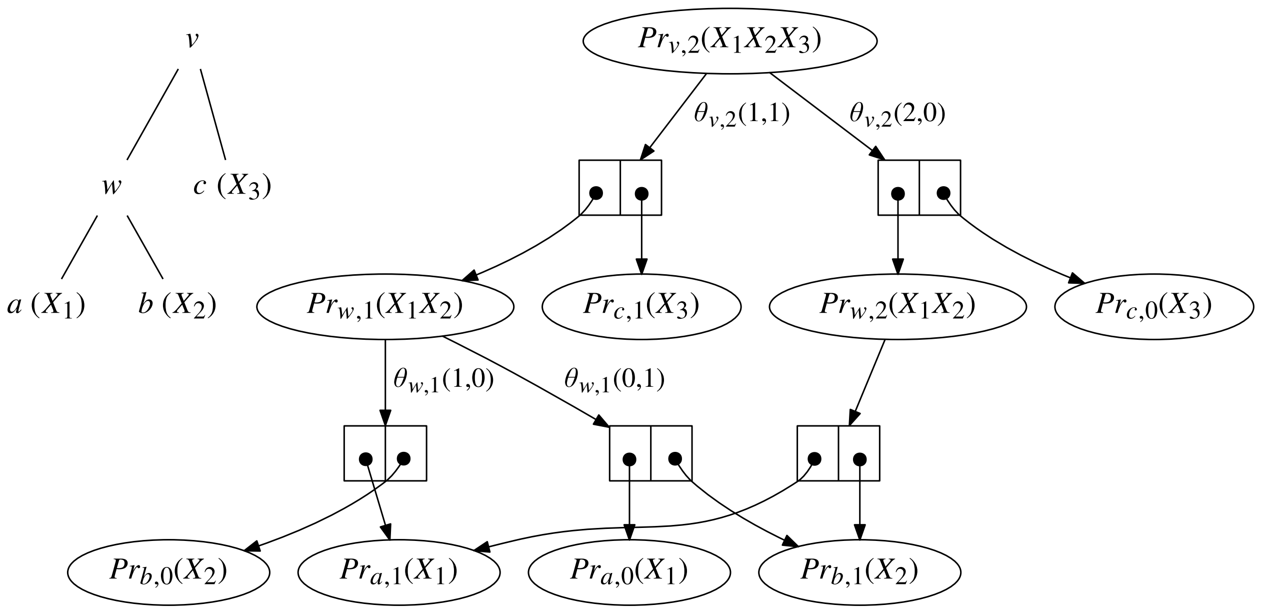


Figure 3.2: A vtree (upper-left), and a corresponding recursive 3-choose-2 distribution (right). Leaf vtree nodes are labeled with their variables inside parenthesis.

tively (hence, $i_1 + i_2 = i$). The underlying independence assumption in the above inductive case is this: how we select i_1 elements from v_1 is independent of how we select i_2 elements from v_2, after we have chosen how many elements i_1 and i_2 to select in each.

Figure 3.2 depicts a vtree and a corresponding 3-choose-2 model. Each circled node represents a recursive n-choose-k model that is associated with an internal vtree node, for particular values of n and k. Each circled node is also associated with a choice distribution, whose parameters annotate the edges outgoing the node.

Example 3. Using the recursive n-choose-k model, and the choice distributions of Example 2, the probability that a student takes machine learning (`ML`), logic (`LG`) and linear algebra (`LA`) is

$$\begin{aligned}
&Pr(\mathtt{ML}=1, \mathtt{LG}=1, \mathtt{CP}=0, \mathtt{CX}=0, \mathtt{LA}=1, \mathtt{CL}=0) \\
&= \theta_{\mathsf{courses},3}(2,1) \cdot \theta_{\mathsf{cs},2}(2,0) \cdot \theta_{\mathsf{math},1}(1,0) \cdot \\
&\qquad \theta_{\mathsf{ai},2}(1,1) \cdot \theta_{\mathsf{theory},0}(0,0) \\
&= 0.3 \cdot 0.1 \cdot 0.3 \cdot 1 \cdot 1 = 0.009.
\end{aligned}$$

□

Finally, we show that our recursive n-choose-k model subsumes the logistic n-choose-k model of Equation 3.1.

Proposition 1. *For any logistic n-choose-k model, there is a recursive n-choose-k model that induces the same distribution.*

3.3.2 Tractable Inference

Recursive n-choose-k models are *tractable* probabilistic models: we can perform many probabilistic queries in time linear in the size of the model. For example, we can compute the most probable explanation (MPE), the probability of evidence, and posterior marginals, all in linear time. For example, we can use MPE to extend a partial subset selection to a complete one (e.g., to extend a user's top-3 list of movies to a top-10 list of movies, to provide movie suggestions). We can perform cardinality queries efficiently: given a user's top-3 list, what is the expected number of comedies that would appear on their top-10 list? This tractability is inherited from the Probabilistic Sentential Decision Diagram [KVC14], of which the recursive n-choose-k model is a concrete example. We discuss this connection further in Section 3.6.

As an example, consider a recursive n-choose-k model and suppose we observed evidence $\mathbf{e}$ on some of its variables $\mathbf{E} \subseteq \mathbf{X}$. We can compute the probability of this evidence recursively, starting from the root vtree node v:

$$Pr_{v,i}(\mathbf{e}) = \sum_{\theta_{v,i}(i_1,i_2)} Pr_{v_1,i_1}(\mathbf{e}_{v_1}) Pr_{v_2,i_2}(\mathbf{e}_{v_2}) \theta_{v,i}(i_1,i_2),$$

which follows from the underlying independence assumptions. In the base case, v is a leaf vtree node with variable X, and $i \in \{0,1\}$. If the evidence $\mathbf{e}$ is empty, then $Pr_{v,i}(\mathbf{e}) = 1$. Otherwise, $Pr_{v_i}(\mathbf{e}) = 1$ iff evidence $\mathbf{e}$ and the 1-choose-i model sets X to the same value.

Example 4. Say we want to compute the probability that a student takes learning (`ML`) and linear algebra (`LA`) out of 3 total classes, with the choice distributions of Example 2. With evidence $\mathbf{e} = \{\texttt{ML} = 1, \texttt{LA} = 1\}$, we have:

$$\begin{aligned}
&Pr_{\textsf{courses},3}(\texttt{ML}{=}1, \texttt{LA}{=}1)\\
&= Pr_{\textsf{cs},2}(\texttt{ML}{=}1) \cdot Pr_{\textsf{math},1}(\texttt{LA}{=}1) \cdot \theta_{\textsf{courses},3}(2,1)\\
&+ Pr_{\textsf{cs},1}(\texttt{ML}{=}1) \cdot Pr_{\textsf{math},2}(\texttt{LA}{=}1) \cdot \theta_{\textsf{courses},3}(1,2)\\
&= Pr_{\textsf{cs},2}(\texttt{ML}{=}1) \cdot 0.3 \cdot 0.3 + Pr_{\textsf{cs},1}(\texttt{ML}{=}1) \cdot 1 \cdot 0.1.
\end{aligned}$$

Recursively, we compute $Pr_{\textsf{cs},2}(\texttt{ML}{=}1) = 0.34$ and $Pr_{\textsf{cs},1}(\texttt{ML}{=}1) = 0.16$, which yields:

$$Pr_{\textsf{courses},3}(\mathbf{e}) = 0.34 \cdot 0.09 + 0.16 \cdot 0.1 = 0.0466.$$ □

3.4 Learning N-Choose-K Models

We show in this section how to estimate the parameters of a recursive n-choose-k model in closed form. We also propose a simple structure learning algorithm for these models, which amounts to learning their underlying vtrees (i.e., the recursive partitioning of variables).

We first consider the number of parameters in a recursive n-choose-k model, which is relevant to our structure learning algorithm. Each leaf vtree node corresponds to a 1-choose-1 or a 1-choose-0 model, which has no parameters. There are $O(n)$ internal nodes in the vtree, and each one has $O(k)$ choice distributions associated with it. Each of these distributions has $O(k)$ parameters, leading to a total of $O(nk^2)$ parameters. Hence, the total number of parameters in a recursive n-choose-k model is bounded by a polynomial in n and k.

To contrast, there are n parameters in a logistic n-choose-k model, which can be learned by iterative methods such as gradient descent [STA12]. Moreover, unlike our recursive model, there is no structure to be learned in a logistic n-choose-k model.

3.4.1 Parameter Learning

Suppose we are given a set of n binary variables $\mathbf{X}$. Let $\mathcal{D}$ be a dataset containing N examples, where each example is an instantiation $\mathbf{x}$ of variables $\mathbf{X}$ with exactly k variables set to true (that is, $\mathcal{D}$ is an n-choose-k dataset).

For a set of variables $\mathbf{Y} \subseteq \mathbf{X}$, we will say that an example $\mathbf{x}$ has $\mathbf{Y}$-cardinality equal to m iff exactly m variables in $\mathbf{Y}$ are set to true in the example. We will also use $\mathcal{D}\#(\mathbf{Y}\!:\!m)$ to denote the number of examples in dataset $\mathcal{D}$ with $\mathbf{Y}$-cardinality equal to m. This allows us to define the following empirical probability, which is the probability of having $\mathbf{Y}$-cardinality equal to m:

$$Pr_{\mathcal{D}}(\mathbf{Y}\!:\!m) = \tfrac{1}{N}\mathcal{D}\#(\mathbf{Y}\!:\!m).$$

Example 5. Consider the example

$$\texttt{ML}\!=\!0, \texttt{LG}\!=\!1, \texttt{CP}\!=\!1, \texttt{CX}\!=\!0, \texttt{CL}\!=\!0, \texttt{LA}\!=\!1$$

which has an ai-cardinality of 1 and a cs-cardinality of 2. We can compute the empirical probability that a student takes one ai course and two cs courses by counting examples in the dataset:

$$Pr_{\mathcal{D}}(\mathsf{ai}\!:\!1, \mathsf{cs}\!:\!2) = \tfrac{1}{N}\mathcal{D}\#(\mathsf{ai}\!:\!1, \mathsf{cs}\!:\!2).$$

We can also find the conditional probability that a student takes one ai course given that they take two cs courses:

$$Pr_{\mathcal{D}}(\mathsf{ai}\!:\!1 \mid \mathsf{cs}\!:\!2) = \frac{\mathcal{D}\#(\mathsf{ai}\!:\!1, \mathsf{cs}\!:\!2)}{\mathcal{D}\#(\mathsf{cs}\!:\!2)}. \qquad \square$$

The following theorem provides a closed form for the maximum likelihood estimates of a recursive n-choose-k model given a corresponding dataset $\mathcal{D}$.

Theorem 1. *Consider a recursive n-choose-k model and dataset $\mathcal{D}$, both over variables $\boldsymbol{X}$. Let v be an internal vtree node of this model. The maximum-likelihood parameter estimates for node v are unique and given by*

$$\begin{aligned}\theta_{v,i}(i_1, i_2) &= Pr_{\mathcal{D}}(\boldsymbol{X}_{v_1}\!:\!i_1 \mid \boldsymbol{X}_v\!:\!i) \\ &= Pr_{\mathcal{D}}(\boldsymbol{X}_{v_2}\!:\!i_2 \mid \boldsymbol{X}_v\!:\!i).\end{aligned}$$

According to this theorem, and given the underlying vtree of a recursive n-choose-k model, we can estimate its maximum likelihood parameters by performing a single pass on the given dataset.

3.4.2 Structure Learning

We now turn to learning the structure of a recursive n-choose-k model, which amounts to learning its underlying vtree. Our approach will be based on maximizing the log-likelihood of the model, without penalizing for structure complexity since the number of parameters of any recursive n-choose-k model is bounded by a polynomial (i.e., regardless of its underlying vtree).

Our approach relies on the following result, which shows that the log-likelihood of a recursive n-choose-k model M, denoted $\mathcal{LL}(M \mid \mathcal{D})$, decomposes over vtree nodes.

Theorem 2. *Consider a recursive n-choose-k model M over variables $\boldsymbol{X}$ and a corresponding dataset $\mathcal{D}$. Let v be an internal vtree node of this model. We then have*

$$\begin{aligned}\mathcal{LL}(M \mid \mathcal{D}) &= -N \cdot \sum_v H(\boldsymbol{X}_{v_1} \mid \boldsymbol{X}_v) \\ &= -N \cdot \sum_v H(\boldsymbol{X}_{v_2} \mid \boldsymbol{X}_v)\end{aligned}$$

where $H(\boldsymbol{X}_{v_1} | \boldsymbol{X}_v)$ is the (empirical) conditional entropy of the cardinality of $\boldsymbol{X}_{v_1}$ given the cardinality of $\boldsymbol{X}_v$:

$$-\sum_{\theta_{v,i}(i_1,i_2)} Pr_{\mathcal{D}}(\boldsymbol{X}_{v_1} \!:\! i_1, \boldsymbol{X}_v \!:\! i) \cdot \log Pr_{\mathcal{D}}(\boldsymbol{X}_{v_1} \!:\! i_1 \mid \boldsymbol{X}_v \!:\! i).$$

Theorem 2 suggests a greedy heuristic for selecting a vtree. We start with n vtrees, each over a single variable $X \in \mathbf{X}$. We greedily pair two vtrees v_a and v_b (i.e., make them children of a new vtree node v) if they have the lowest conditional entropy $H(\mathbf{X}_{v_a} \mid \mathbf{X}_v) = H(\mathbf{X}_{v_b} \mid \mathbf{X}_v)$ over all pairs v_a and v_b. We iterate until we have a single vtree over all variables $\mathbf{X}$.

Example 6. Figure 3.3 highlights the first few iterations of our vtree learning algorithm, using our course selection example. Initially, at iteration $i = 0$, we have six vtrees, one for each of the six courses. Over all pairs of vtrees, say we obtain the lowest conditional entropy with $H(\{\texttt{LG}\}|\{\texttt{LG}, \texttt{ML}\})$. At iteration $i = 1$, we pair the vtrees of `ML` and `LG` to form a new vtree over both. Over all pairs of vtrees, say we now obtain the lowest conditional entropy with $H(\{\texttt{CX}\}|\{\texttt{CX}, \texttt{CP}\})$. At iteration $i = 2$, we again pair the corresponding vtrees to form a new one. We repeat, until we obtain a single vtree. □

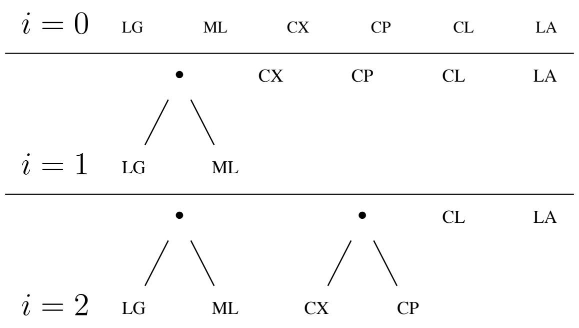


Figure 3.3: The first few iterations of vtree learning.

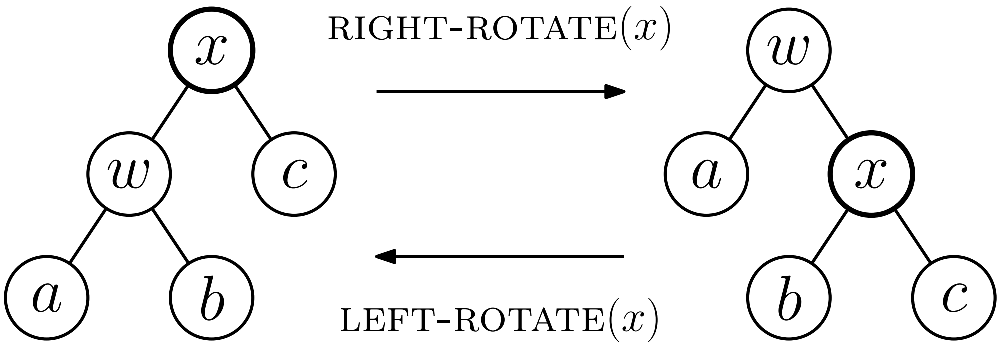


Figure 3.4: Rotating a vtree node x right and left. Nodes a, b, and c may represent leaves or subtrees.

Our structure learning algorithm improves the quality of this vtree using local search (simulated annealing in particular). To navigate the full space of vtrees, it suffices to have (left and right) tree rotation operators, and an operator to swap the labels of two vtree leaves.[1] Figure 3.4 highlights the rotation operator for vtrees. In our experiments, we used simulated annealing with the above operators to navigate the search space of vtrees, using the greedily found vtree we just described as the initial vtree.

Our local search algorithm defines two vtrees as neighbors if one can be obtained from the other by performing a left/right rotation of an internal vtree node, or by swapping the variables of two leaf vtree nodes. We used an exponential cooling schedule for simulated annealing. That is, during each iteration of the algorithm, we first select a neighbor at

[1][CD13] used rotation operators and an operation that swapped the children of an internal vtree node in order to navigate the space of vtrees. In our recursive n-choose-k model, the log likelihood is invariant to the swapping operator, hence we chose to swap the labels of leaf nodes.

random. If it has a better log-likelihood score, we move to that neighbor. Otherwise, we move to that neighbor with probability $\exp\{\frac{1}{T_i}\Delta\mathcal{LL}(M \mid \mathcal{D})\}$, where $\Delta\mathcal{LL}(M \mid \mathcal{D})$ is the difference in log-likelihood, and T_i is the temperature at the current iteration. We stop when the temperature drops to a preset threshold.[2] We do not use restarts (empirically, our greedy heuristic appears to obtain a reasonably good initial vtree).

Finally, we remark on the simplicity of our structure learning algorithm, compared to other tractable representations such as the arithmetic circuit (AC) and their sum-product network (SPN) variant [CD17]. In Section 3.6, we discuss how our recursive n-choose-k model corresponds to a certain type of AC. However, rather than search for ACs [LD08, GD13, DV15], we only need to search for vtrees (a much simpler space). This is possible due to a property called canonicity, which fixes the structure of the AC once the vtree is fixed [KVC14].

3.5 Experiments And Case Studies

We next compare our recursive n-choose-k model with the logistic n-choose-k model of [STA12], using simulated n-choose-k datasets. We later evaluate these models using real-world datasets from the domains of preference learning and sports analytics.

3.5.1 Simulated Data

Based on Proposition 1, for a given logistic n-choose-k model, there exists a parameterization of a recursive n-choose-k model that induces the same distribution. However, the logistic n-choose-k model has fewer parameters, as discussed before. Thus, for less data we generally expect the logistic version to be more robust to overfitting, and for greater amounts of data we generally expect our recursive version to ultimately provide a better fit.

The first goal in our experiments is to verify this behavior. We simulated n-choose-

[2]In our experiments, we have an initial temperature of 5.0, a cooling rate of 0.98, and a temperature threshold of 0.001.

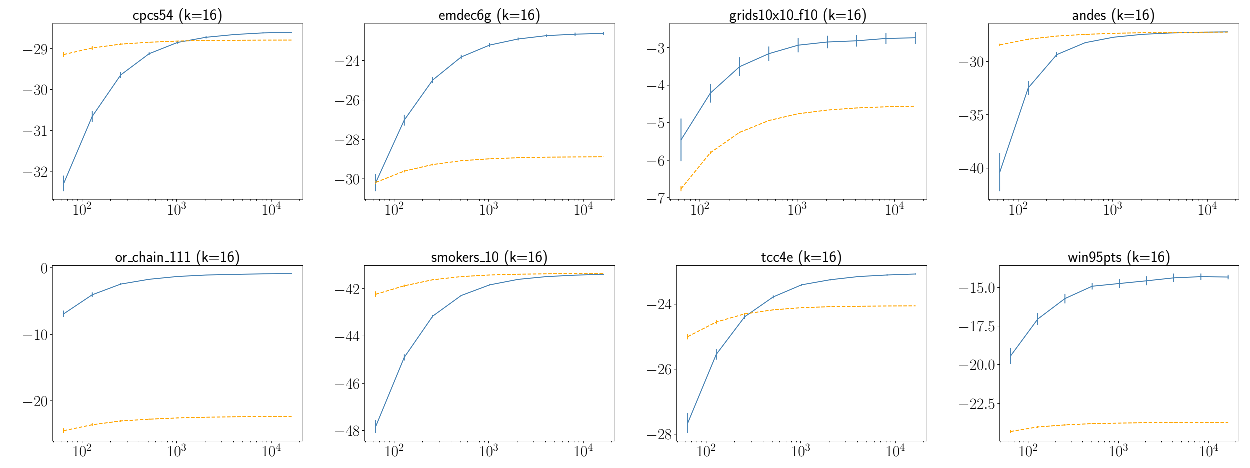


Figure 3.5: Learning results for cardinality-16: dataset size (x-axis) vs test log likelihood (y-axis). The blue solid lines and orange dashed lines correspond to the recursive and logistic n-choose-k models, respectively.

k datasets, that are independent of both the logistic and recursive n-choose-k models (so that neither model would be able to fit the data perfectly). In particular, we simulated datasets from Bayesian and Markov networks, but subjected the networks to cardinality-k constraints.[3]

We selected a variety of networks from the literature, over binary variables (the corresponding variable count is given in parentheses): `cpcs54` (54), and `win95pts` (76) are classical diagnostic BNs from the literature; `emdec6g`(168) and `tcc4e` (98) are noisy-or diagnostic BNs from HRL Laboratories; `andes` (223) is a Bayesian network for educational assessment; `grid10x10_f10` (100), `or_chain_111` (200), `smokers_10` (120) are networks taken from previously held UAI competitions. We first considered cardinality-16 datasets. For each, we simulated a testing set of size 2,500 and independently sampled 20 training sets each of size 2^s for s from 6 (64 examples) to 14 (16,384 examples). We trained n-choose-k models from each training set, and evaluated them using the testing set, in Figure 3.5. Each point in a plot is an average over 20 training sets.

[3]We remark that it is non-trivial to simulate a Bayesian network, when we condition on logical constraints. To do so, efficiently, we first compiled a Bayesian network to a PSDD, and then multiplied it with a (uniform) PSDD representing a cardinality-k constraint, which is discussed in Chapter 5. The result is a PSDD, which we can now efficiently sample from.

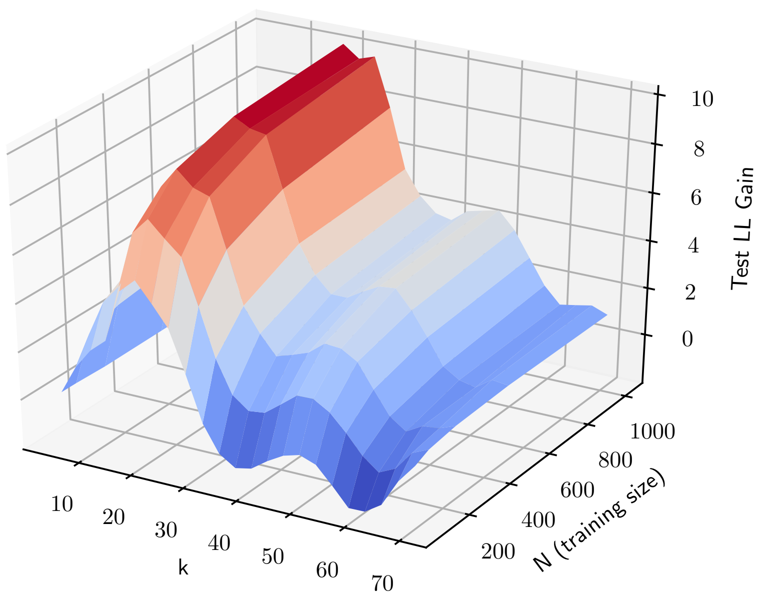


Figure 3.6: Learning results on the `win95pts` dataset: k vs dataset size vs test log likelihood.

Our recursive n-choose-k model is depicted with a solid blue line, and the logistic n-choose-k model is depicted with a dashed orange line. There are a few general trends. First, the logistic n-choose-k model more often provides a better fit with smaller amounts of data, but in all but two cases (`smokers_10` and `andes`), our recursive alternative will eventually learn a better model given enough data. Finally, we note that the variance (plotted using error-bars) of the logistic n-choose-k is smaller (since it has fewer parameters).

Figure 3.6 highlights the impact of varying k, using data simulated from the `win95pts` network. Here, observe the gain obtained from using the recursive model versus the logistic model, in terms of the test log likelihood, i.e., $\mathcal{LL}_{\text{recursive}}(\mathcal{D}) - \mathcal{LL}_{\text{logistic}}(\mathcal{D})$. Hence, if the gain is negative, the logistic model obtained a better likelihood, and if the gain is positive, then our recursive model obtained a better likelihood. Again, as we increase the size of the dataset, our recursive n-choose-k model obtains better likelihoods. As we vary the cardinality of the examples in the data, we see that the performance can vary. Generally, as we increase k from 0 to n, the difference between the models become greater up to a point and then it decreases again. This is expected to an extent since there is an underlying symmetry: a constraint that k values be positive is equivalent to a constraint that $n-k$ values are negative. Hence, for a given n-choose-k model, there is an equivalent n-choose-$(n-k)$ model where the signs have been switched. However, there is not a perfect symmetry in the distribution, since the original distribution generating the data (`win95pts` in this case) does not have this symmetry across cardinalities.

Finally, we remark on the running time of structure learning. On the `win95pts` network, with a 76-choose-32 dataset of size 2^{13}, our structure learning algorithm runs for only 65.02s (on average over 20 runs). On the `andes` network, with a 223-choose-32 dataset of size 2^{13}, it runs for only 367.8s (on average over 20 runs). This is in contrast to other structure learning algorithms for tractable models, such as those based on ACs and their SPNs variant, where learning requires hours for comparably sized learning problems, and even days and weeks for larger scale problems; see, e.g., [RKG14] for a discussion. While our recursive n-choose-k model corresponds to a special class of ACs (as we shall discuss in Section 3.6), it suffices to learn a vtree and not the AC itself.

3.5.2 Case Study: Preference Learning

We consider a case study in preference learning. The `sushi` dataset consists of $5,000$ total rankings of 10 different types of sushi [Kam03]. From this dataset over total rankings, we can obtain a dataset over top-5 preferred sushi, where we have 10 variables (one for each type of sushi) and a variable is true iff they are in the top-5 of their total ranking; we thus ignore the specific rankings. Note that the resulting dataset is complete, with respect to top-5 rankings. We learned a 10-choose-5 model from this data, using simulated annealing seeded with our conditional entropy heuristic.[4]

Figure 3.7 highlights the learned vtree structure, which we can view as providing a recursive partitioning to guide a selection of the top-5 preferred sushi, as in Section 3.3. Going roughly left-to-right, we start with 3 popular non-fish types of sushi: squid, shrimp and sea eel. Next, egg and cucumber roll are relatively not preferred; next, fatty tuna is heavily preferred. Finally, we observe salmon roe and sea urchin (which are both considered acquired

[4]The sushi data was split into a training set of size $3,500$ and a testing set of size $1,500$ as in [LB11]. Our model was learning using just the training set.

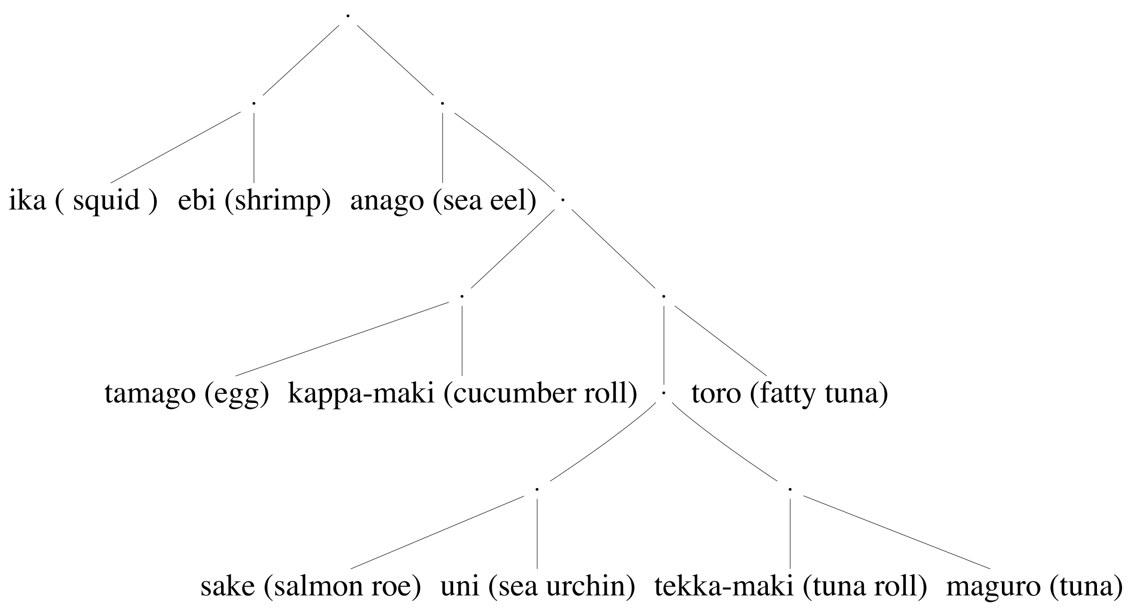


Figure 3.7: 10-choose-5 model for the `sushi` dataset.

tastes) and then tuna roll and tuna. These observations are consistent with previously made observations about the `sushi` dataset; see, e.g., [LB11, CVD15]. In contrast, [LB11] learned a mixture-of-Mallows model with 6 components, providing 6 different reference rankings (and a dispersion parameter).[5] [CVD15] learned a PSDD, but without learning a vtree; a fixed vtree was used based on rankings, which does not reflect any relationships between different types of sushi.

3.5.3 Case Study: Sports Analytics

Team sports, such as basketball and soccer, have an inherent n-choose-k problem, where a coach has to select k out of n players to fulfill different roles on a team. For example, in modern basketball, a team consists of five players who play different roles: there are two guards, who are passers and long-distance shooters; there are two forwards, who generally play closer to the basket; and there is the center, who is normally the tallest player and is the one mostly responsible for gathering rebounds and for contesting shots.

[5]The Mallows model [Mal57] is a probabilistic model for ranking, which assumes a *reference* ranking σ, with other rankings σ' becoming less likely as their distance from σ increases.

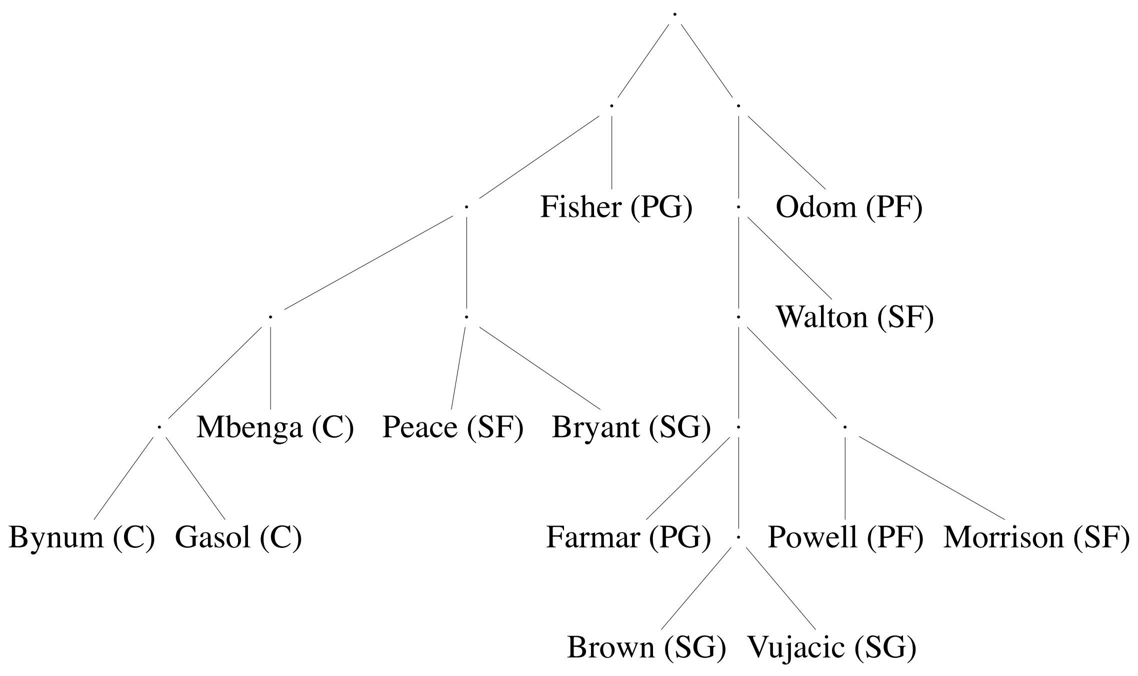


Figure 3.8: 13-choose-5 model for the 2009-2010 Lakers.

`http://stats.nba.com` provides, for a given season and team, a record of all lineups of five players that played together at the same time, and for how many minutes they played. There are 48 minutes played in a basketball game, and 82 games played during the regular season, for a total $3,936$ minutes. For the 2009-2010 Los Angeles Lakers, that season's NBA champions, we obtained a 13-choose-5 dataset with $39,360$ examples, taking lineups in $\frac{1}{10}$-th minute increments, plus some additional examples due to overtime.

Figure 3.8 highlights the (vtree) structure that we learned from the full dataset. Again, we can view this structure as providing a recursive partitioning to guide our selection of a five-player NBA lineup, as in Section 3.3. Starting from the left, and rotating around the root: Andrew Bynum, Pao Gasol and Didier Ilunga-Mbenga are the centers; Metta World Peace, Kobe Bryant, Derek Fisher, and Lamar Odom are the starting non-centers; Luke Walton, Josh Powell and Adam Morrison are the reserve forwards; and Jordan Farmar, Shannon Brown, and Sasha Vujacic are the reserve guards.

3.6 Discovering the Recursive N-Choose-K Model

We now highlight how the recursive n-choose-k model was *discovered* using the Probabilistic Sentential Decision Diagram (PSDD) [KVC14].

A PSDD allows one to define a probability distribution over a *structured probability space,* which is a subset of the Cartesian product of a set of variables—with each element of this subset corresponding to some object of interest. For example, a structured probability space may correspond to the space of permutations, partial rankings or routes on a map [CVD15, CTD16]. PSDDs over structured spaces are interpretable in a precise sense. For certain spaces, including those for subset selection, this interpretability may lead to models whose semantics are so clear that they can be described independently, without the need to invoke the notion of a PSDD in the first place. In such cases, we say that the PSDD has enabled *model discovery.*

Underlying the PSDD is the Sentential Decision Diagram (SDD), which is a class of tractable Boolean circuits [Dar11]. In this framework, the SDD circuit is used to define the structured probability space (a variable instantiation belongs to the structured space iff the SDD circuit outputs one under that instantiation). Once the structured space is defined by an SDD, a PSDD is used to induce a distribution over that space (the PSDD is basically a parameterized SDD).

Consider the recursive 3-choose-2 model of Figure 3.2. This model corresponds to an SDD circuit when we replace (1) internal circled nodes with or-gates, (2) paired boxes with and-gates, (3) 1-choose-1 leaf nodes with a positive literal X and 1-choose-0 leaf nodes with a negative literal $\neg X$. The result is an SDD circuit whose satisfying instantiations have exactly two positive literals; that is, the structured probability space for 3-choose-2. Section 3.7 provides an example of this SDD circuit, and its annotation as a PSDD.

More generally, the following theorem describes SDD circuits whose satisfying instantiations are those with exactly k variables set to true (SDDs are also constructed based on vtrees; see [Dar11] for details).

Proposition 2. *Let $f_{v,k}$ be an SDD circuit, for a vtree v with n variables $\boldsymbol{X}$, whose satisfying* *instantiations $\boldsymbol{x}$ set exactly k variables to true. This circuit is equivalent to:*

$$\bigvee_{k_1+k_2=k} f_{v_1,k_1} \wedge f_{v_2,k_2}$$

where $0 \leq k_1 \leq |\boldsymbol{X}_{v_1}|$ *and* $0 \leq k_2 \leq |\boldsymbol{X}_{v_2}|$.

Hence, each or-gate of an SDD corresponds to a Boolean formula representing an n-choose-k constraint.

To emphasize the clear semantics of this SDD, consider the number of satisfying instantiations (i.e., model count) that an n-choose-k constraint has: $\binom{n}{k}$. To obtain the model count of an SDD (i.e., the number of satisfying instantiations), we replace each or-gate with a $+$ and each and-gate with a $*$, and all literals with a 1. We then evaluate the circuit bottom-up to evaluate the model count. The model count of the SDD in Figure 3.2 represents the following computation of $\binom{3}{2}$:

$$\binom{3}{2} = \binom{2}{1}\binom{1}{1} + \binom{2}{2}\binom{1}{0} = 2 \cdot 1 + 1 \cdot 1 = 3.$$

For a more general example, suppose we are given a vtree over a set of n variables $\mathbf{X}$, where the left child of each internal node is a leaf (this is called a right-linear vtree). Computing the model count of an SDD for this vtree, as shown above, yields the well-known recurrence for binomial coefficients:

$$\binom{n}{k} = \binom{1}{0}\binom{n-1}{k} + \binom{1}{1}\binom{n-1}{k-1} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

To obtain a PSDD from an SDD, one assigns a local distribution on the inputs of each or-gate [KVC14]. For the SDDs of Proposition 2, these local distributions correspond to the choice distributions of our recursive n-choose-k model; see Section 3.7 for an example. This observation allowed us to describe this model in a manner independent of the PSDD framework, and hence enabled *model discovery*.

3.7 Example PSDD

Figure 3.9 highlights the SDD/PSDD corresponding to the recursive 3-choose-2 model of Figure 3.2 using the same vtree.

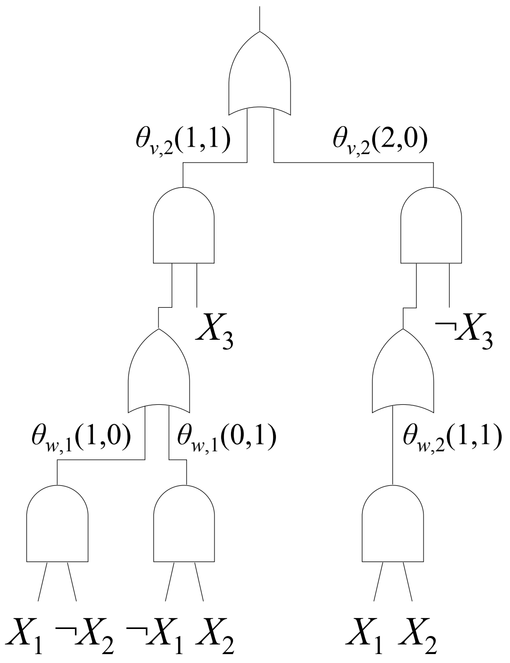


Figure 3.9: A PSDD corresponding to the vtree and the recursive n-choose-k model of Figure 3.2.

As we highlighted in Section 3.6, the Boolean circuit of Figure 3.9 (ignoring the annotated parameters $\theta_{v,k}$) outputs 1 if the circuit input sets exactly 2 out of 3 variables positively, and outputs 0 otherwise. Note that for simplicity, we have omitted inconsistent branches of or-gates that would normally appear in a SDD/PSDD (these branches correspond to instantiations that do not have the required cardinality, and hence, always outputs a 0).

We can obtain an AC of this PSDD by performing two steps: convert each and-gate into a $*$-node, and convert each or-node with children $c_1, \ldots, c_n$ and parameters $\theta_1, \ldots, \theta_n$ into a $+$-node with children $\alpha_1 * c_1, \ldots, \alpha_n * c_n$. Given an instantiation $\mathbf{x}$, the output of the AC is found by setting the inputs to 1/0 according to $\mathbf{x}$ and then evaluating the circuit bottom-up. This output yields the probability $Pr(\mathbf{x})$ of the corresponding recursive 3-choose-2 model.

The properties of SDDs and PSDDs allow certain queries or operations to be performed efficiently, which are otherwise hard on general Boolean and arithmetic circuits. For example, model counting can be performed using SDDs in time that is linear in the size of the SDD [Dar11]. In PSDDs, queries such as MPE and marginals are similarly tractable (as discussed in Section 3.3.2). The maximum likelihood parameters of a PSDD can be learned in closed-from from a complete dataset (as in Section 3.4). Further, one can multiply two PSDDs in polynomial time, which enables incremental learning and inference that is shown in Chapter 5.

3.8 Conclusion

We proposed in this chapter the recursive n-choose-k model for subset selection problems. We also derived a closed-form parameter estimation algorithm for these models, and a simple structure learning algorithm based on greedy and local search. Empirically, we showed how our recursive n-choose-k models can obtain better fits of the data, compared to a previously proposed model. Moreover, we showed how structure search can lead to an intuitive generative model of the subset selection process (based on vtrees). We finally showed how the proposed model was discovered using the PSDD representation for inducing distributions over structured spaces, with the structured space being the set of variable instantiations having a fixed cardinality.

3.A Proofs

To prove Proposition 1, we reduce the logistic n-choose-k model to a weighted model counting (WMC) problem.

Given a propositional sentence Δ and a set of weights $W(\ell)$ on each literal ℓ, its weighted model count is

$$WMC(\Delta) = \sum_{\mathbf{x} \models \Delta} W(\mathbf{x}) = \sum_{\mathbf{x} \models \Delta} \prod_{\mathbf{x} \models \ell} W(\ell)$$

where the weight of a model $W(\mathbf{x})$ is the product of the weights of its literals $W(\ell)$. For more on weighted model counting see, e.g., [CD08, KVD17].

A WMC problem induces a distribution over its models:

$$Pr(\mathbf{x}) = \frac{W(\mathbf{x})}{WMC(\Delta)}.$$

If a sentence Δ can be compiled into an SDD, then the SDD can be used to compute its weighted model count. Subsequently, a PSDD can represent the corresponding distribution, as follows.

Lemma 1. *Consider a WMC problem over a propositional sentence Δ with weights $W(\ell)$ on each literal ℓ. Let m be an SDD representing sentence Δ. There is a PSDD with m as its base that induces the same distribution induced by the given WMC problem.*

Proof. Given a normalized SDD for Δ, we show how to parameterize it as a PSDD. For an SDD/PSDD node m, let P_m be the distribution induced by the WMC problem on m, and let Q_m be the distribution induced by the PSDD. If m is a terminal node, set $Q_m(\ell) = \eta \cdot W(\ell)$ if ℓ is compatible with the base of m and 0 otherwise, where η is a normalizing constant so that Q_m sums to one. If m is a decision node with elements (p_i, s_i, θ_i), set

$$\theta_i = \frac{WMC(p_i) \cdot WMC(s_i)}{WMC(m)}.$$

We show $P_m(\mathbf{x}) = Q_m(\mathbf{x})$ for all $\mathbf{x}$, by induction. The base case, where m is a terminal node, is immediate. Suppose m is a decision node with elements (p_i, s_i, θ_i) with prime variables $\mathbf{X}$ and sub variables $\mathbf{Y}$, and where $P_{p_i}(\mathbf{x}) = Q_{p_i}(\mathbf{x})$ and $P_{s_i}(\mathbf{y}) = Q_{s_i}(\mathbf{y})$. Given an assignment $\mathbf{xy}$, let the i-th element (p_i, s_i, θ_i) be the one where $\mathbf{x} \models p_i$. We have:

$$\begin{aligned}
Q_m(\mathbf{xy}) &= Q_{p_i}(\mathbf{x}) \cdot Q_{s_i}(\mathbf{y}) \cdot \theta_i \\
&= P_{p_i}(\mathbf{x}) \cdot P_{s_i}(\mathbf{y}) \cdot \theta_i && \text{by induction} \\
&= \frac{W(\mathbf{x})}{WMC(p_i)} \cdot \frac{W(\mathbf{y})}{WMC(s_i)} \cdot \frac{WMC(p_i) \cdot WMC(s_i)}{WMC(m)} \\
&= \frac{W(\mathbf{x}) \cdot W(\mathbf{y})}{WMC(m)} = \frac{W(\mathbf{xy})}{WMC(m)} = P_m(\mathbf{xy}). && \square
\end{aligned}$$

$\square$

Proof of Proposition 1. We can represent the logistic n-choose-k model of Equation 3.1 as a weighted model counting problem problem. First, let Δ be a logical n-choose-k constraint as in Proposition 2. If we use the weights $W(X) = \exp\{\theta_X\}$ and $W(\neg X) = 1$, then the weighted model count gives us the partition function of the logistic n-choose-k model of Equation 3.1.

Using Proposition 2, we can obtain an SDD for Δ of polynomial size. Using the construction of Lemma 1, we obtain a PSDD that corresponds to a recursive n-choose-k model. This distribution is equivalent to the one induced by the WMC problem, and the one induced by the given logistic n-choose-k model. $\square$

Proof of Theorem 1. Under the recursive n-select-k distribution, the probability $Pr_{w,k}(\mathbf{x})$ is a product of $n-1$ choice parameters. Hence, the log likelihood decomposes as follows:

$$\begin{aligned}
&\mathcal{LL}(M \mid \mathcal{D}) = \sum_{a=1}^{N} \log Pr_{w,k}(\mathbf{x}) \\
&= \sum_{v,i} \sum_{\theta_{v,i}(i_1,i_2)} \mathcal{D}\#(\mathbf{X}_{v_1}\!:\!i_1, \mathbf{X}_v\!:\!i) \log \theta_{v,i}(i_1,i_2) \\
&= N \cdot \sum_{v,i} \sum_{\theta_{v,i}(i_1,i_2)} Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1, \mathbf{X}_v\!:\!i) \log \theta_{v,i}(i_1,i_2)
\end{aligned}$$

Note that for each v and i, all of the local choice distributions $\theta_{v,i}$ are independent. Hence it suffices to locally maximize each component:

$$\sum_{\theta_{v,i}(i_1,i_2)} Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1, \mathbf{X}_v\!:\!i) \log \theta_{v,i}(i_1,i_2)$$

which is basically a cross entropy that is maximized at:

$$\begin{aligned}
\theta_{v,i}(i_1,i_2) &= Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1 \mid \mathbf{X}_v\!:\!i) \\
&= Pr_{\mathcal{D}}(\mathbf{X}_{v_2}\!:\!i_2 \mid \mathbf{X}_v\!:\!i).
\end{aligned}$$

$\square$

$\square$

Proof of Theorem 2. If we substitute the maximum likelihood estimates of Theorem 1 into the log likelihood of an n-choose-k model we obtain our result.

First, consider the component contributed by a single vtree node v and their choice distribution $\theta_{v,i}$:

$$\begin{aligned}
&N \sum_{\theta_{v,i}(i_1,i_2)} Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1, \mathbf{X}_v\!:\!i) \log \theta_{v,i}(i_1,i_2) \\
&= N \sum_{\theta_{v,i}(i_1,i_2)} Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1, \mathbf{X}_v\!:\!i) \log Pr_{\mathcal{D}}(\mathbf{X}_{v_1}\!:\!i_1 | \mathbf{X}_v\!:\!i) \\
&= -N \cdot H(\mathbf{X}_{v_1}\!:\!i_1 | \mathbf{X}_v\!:\!i)
\end{aligned}$$

which is the conditional entropy distribution. Hence:

$$\mathcal{LL}(M \mid \mathcal{D}) = -N \sum_{v} H(\mathbf{X}_{v_1} \!:\! i_1 | \mathbf{X}_v \!:\! i) \qquad \square$$

$\square$

Proof of Proposition 2. Consider an n-choose-k constraint $f_{v,k}$ associated with a vtree node v, with children v_1 and v_2 over variables $\mathbf{X}_{v_1}$ and $\mathbf{X}_{v_2}$.

An $\mathbf{X}_{v_1}$-$\mathbf{X}_{v_2}$ decomposition is found by compressing the decomposition:

$$f_{v,k} = \bigvee_{\mathbf{x}_{v_1}} \mathbf{x}_{v_1} \wedge f_{v,k}|\mathbf{x}_{v_1}$$

which is found by disjoining all $\mathbf{x}_{v_1}$ terms that have equivalent terms $f_{v,k}|\mathbf{x}_{v_1}$. For all $\mathbf{x}_{v_1}$ with the same cardinality k_1, the resulting function $f_{v,k}|\mathbf{x}_{v_1}$ is the same. When we disjoin all such $\mathbf{x}_{v_1}$ we obtain the function f_{v_1,k_1}. Further, $f_{v,k}|\mathbf{x}_{v_1} = f_{v_2,k_2}$ for $k_2 = k - k_1$. Hence, the compressed decomposition is:

$$f_{v,k} = \bigvee_{k_1+k_2=k} f_{v_1,k_1} \wedge f_{v_2,k_2}.$$

See also [MT98, Weg00] (such as the cardinality-k constraint) for more on symmetric functions on OBDDs. $\square$

CHAPTER 4

Structured Bayesian Networks

In this chapter, we first propose a variant on PSDDs, called *conditional PSDDs,* for representing a family of distributions that are conditioned on the same set of variables. Later, we use conditional PSDDs to define a new graphical model, called *structured Bayesian networks,* in which nodes can have an exponential number of states, hence expanding the scope of domains where Bayesian networks can be applied. The results discussed in this chapter appeared in [SCD18].

4.1 Introduction

The Probabilistic Sentential Decision Diagram (PSDD) is a recently proposed tractable representation for probability distributions [KVC14]. PSDDs were motivated by the need to learn distributions in the presence of abundant background knowledge, expressed using Boolean constraints. For example, PSDDs have previously been leveraged to learn distributions in domains that can give rise to massive Boolean constraints, such as user preference rankings [CVD15] as well as modeling games [CTD16]. The typical approach for constructing a PSDD is to first *compile* Boolean constraints into a tractable logical representation called the Sentential Decision Diagram (SDD). A PSDD is then *learned* from the compiled SDD and a given dataset.

While PSDDs can be quite effective in the presence of massive Boolean constraints, they do not allow users to explicitly represent background knowledge in the form of conditional independence constraints. In contrast, probabilistic graphical models use graphs to represent such knowledge [Dar09, KF09, Mur12, Bar12]. Moreover, Bayesian networks represent such knowledge while remaining modular in a sense that we shall explain next and exploit later.

A Bayesian network has two components. The first is a directed acyclic graph (DAG) with its nodes representing variables of interest, and its topology encoding conditional independence constraints. The second component consists of a set of conditional probability tables (CPTs), one for each variable in the network. The CPT for a variable defines a set of distributions for that variable, conditioned on the states of its parents in the network. A key property of Bayesian networks is their modularity, which allows the parameter estimation problem under complete data to be decomposed into local CPT estimation problems, with closed-form solutions. However, in the presence of Boolean constraints, Bayesian networks may have very connected topologies, in addition to variables with many states and parents, potentially making them unusable practically.

Bayesian networks and PSDDs can then be viewed as two extremes on a spectrum. On the one hand, Bayesian networks can exploit background knowledge in the form of conditional independencies, but they cannot handle Boolean constraints as effectively as PSDDs. On the other hand, PSDDs can effectively incorporate background knowledge in the form of Boolean constraints, but cannot directly exploit known conditional independencies as do Bayesian networks.

In this chapter, we propose a representation that inherits advantages from both representations. First, we propose the *conditional PSDD*, which is a tractable representation of probability distributions that are conditioned on the same set of variables. We then use these PSDDs to represent the conditional probability tables (CPTs) of a Bayesian network. This allows us to inherit the modularity of Bayesian networks and their ability to explicitly encode independence, while also inheriting from PSDDs their ability to effectively incorporate Boolean domain constraints. We refer to the resulting representation as a *structured Bayesian network,* as it also allows nodes with (exponentially) many states. This increases the reach of both PSDDs and Bayesian networks, on the modeling, learning and computational fronts.

This chapter is organized as follows. We first review background knowledge as exploited

L	K	P	A	Students
0	0	1	0	6
0	0	1	1	54
0	1	1	1	10
1	0	0	0	5
1	0	1	0	1
1	0	1	1	0
1	1	0	0	13
1	1	1	0	8
1	1	1	1	3

Table 4.1: Student enrollment data. Each column (variable) corresponds to a course, and each row (variable assignment) corresponds to an example. The counts represent the number of times that the example appeared in the dataset. For example, the second row represents those students who took Probability (P) and AI (A), but did not take Logic (L) or KR (K). There were 54 such examples (students) in this case.

by PSDDs. We next consider *modular* forms of background knowledge, which are required and exploited by conditional PSDDs. We subsequently introduce the syntax and semantics of conditional PSDDs, while proposing a simple and efficient algorithm for learning their parameters from complete data. We then provide some empirical results to highlight the statistical advantages of conditional PSDDs and structured Bayesian networks, followed by an example structured Bayesian network that can be used for different prediction tasks. We finally close with some concluding remarks.

4.2 Learning with Background Knowledge

A common form of background knowledge is Boolean constraints, which we illustrate next with several examples. The first example we discuss is due to [KVC14], and concerns a computer science department that organizes four courses: Logic (L), Knowledge Representation

(K), Probability (P), and Artificial Intelligence (A). The department has data on student enrollments, as in Table 4.1, and wishes to learn a probabilistic model of student preferences. For example, the department may use the model to infer whether students who take KR are more likely to take Logic than AI. This is a classical machine learning problem, except that we also have background knowledge in the form of program requirements and prerequisites:

– A student must take at least one of Probability or Logic.

– Probability is a prerequisite for AI.

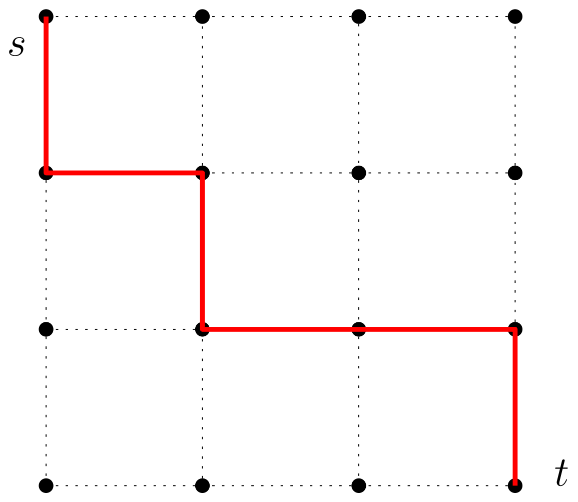
– The prerequisite for KR is either AI or Logic.

Our goal is then to learn a model using both the data in Table 4.1 and the above knowledge. Effectively, what this knowledge tells us is that some examples will never appear in the dataset because they violate domain constraints — in contrast, for example, to being unlikely or missing for some idiosyncratic reason. This is valuable information and ignoring it can lead to learning a suboptimal model at best, as discussed in [KVC14]. More precisely, the domain constraints of this example can be expressed as follows:

$$\begin{aligned} P &\vee L \\ A &\Rightarrow P \\ K &\Rightarrow A \vee L \end{aligned}$$

Even though there are 16 combinations of courses, this knowledge tells us that only 9 of them are valid choices. Hence, an approach that observes this information must learn a probability distribution that assigns a zero probability to every combination that violates these constraints; otherwise, it will be suboptimal.

Consider now another example, where the goal is to learn a distribution over routes on a map. Figure 4.1 depicts two example routes on a map that has the form of a grid [CTD16] — more generally, a map is modeled using an undirected graph. We can represent each edge i in the map by a Boolean variable E_i, and each route as a variable assignment that sets only its corresponding edge variables to true. In this case, each example in the dataset will be a truth assignment to edge variables E_i. Again, some examples will never appear in the dataset



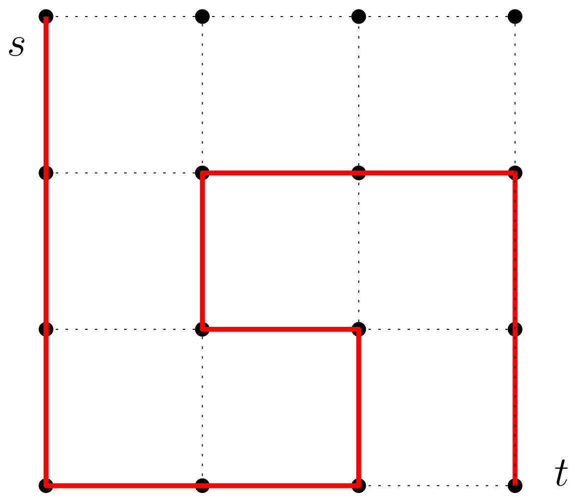


Figure 4.1: Routes on a 4×4 grid.

as they do not correspond to valid routes. For example, a route that contains disconnected edges is invalid, but other types of invalid routes may also be mandated by the domain. That is, we may already know that only simple routes are possible (i.e., no cycles), or that routes which include edge i will never include some other edge j, and so on. Again, the goal here is to be able to learn from both the dataset and the domain constraints; see [CTD16, NYM17] for how connected-routes and simple-routes can be encoded using Boolean constraints.

We now consider our last illustrative example in which we want to learn a distribution to reason about user preferences. We have n items in this case and a corresponding number of ranks: $1, \ldots, n$. Users are asked to specify the rank j of each item i, which can be represented by a Boolean variable A_{ij}. That is, this variable is true if and only if item i has rank j. Each example in this case is also a variable assignment, which declares the ranks of some items. As is, an example could leave some item i unassigned to a rank, i.e., when variables A_{ij} are false for all ranks j. Moreover, an example could assign the same rank to multiple items, i.e., when both A_{ij} and A_{kj} are true for items $i \neq k$ and rank j.

All kinds of constraints may arise in this domain of preference learning [LB11, HKG12]. For example, one may know up front that all examples must correspond to total rankings, in which each item is assigned precisely one rank, and each rank is assumed by exactly one item. These constraints were considered in [CVD15], which showed how to encode them using Boolean constraints. For example, when $n = 3$, the constraints are as follows:

– Each item i is assigned exactly one rank, leading to three constraints for $i \in \{1, 2, 3\}$: $(A_{i1} \wedge \neg A_{i2} \wedge \neg A_{i3}) \vee (\neg A_{i1} \wedge A_{i2} \wedge \neg A_{i3}) \vee (\neg A_{i1} \wedge \neg A_{i2} \wedge A_{i3})$.

– Each rank j is assumed by exactly one item, leading to three constraints for $j \in \{1, 2, 3\}$: $(A_{1j} \wedge \neg A_{2j} \wedge \neg A_{3j}) \vee (\neg A_{1j} \wedge A_{2j} \wedge \neg A_{3j}) \vee (\neg A_{1j} \wedge \neg A_{2j} \wedge A_{3j})$.

More generally, one needs a total of $2n$ constraints when considering n items and ranks.

As discussed in [KVC14], learning from both data and domain constraints is challenging for classical learning approaches. For example, when using a probabilistic graphical model, the resulting graph will almost be fully connected which makes both learning and inference very difficult. Hence, PSDDs were introduced to address this challenge.

4.3 Modular Background Knowledge

The PSDD framework can work with background knowledge in the form of arbitrary Boolean constraints since SDD circuits can be compiled from such constraints [Dar11]. In some cases, however, Boolean constraints may be *modular* in a sense that we shall define precisely in this section. Modular Boolean constraints are important because they can be easily integrated with background knowledge in the form of independence constraints. Moreover, they can facilitate the compilation process into SDDs and the learning process itself, leading to improved scalability and to more accurate models. We define modular Boolean constraints next, starting with a motivation from Bayesian networks.

As mentioned earlier, the first component of a Bayesian network is a directed acyclic graph (DAG) over variables $X_1, \ldots, X_n$, which encodes conditional independence statements. In particular, for each variable X_i with parents $\mathbf{P}_i$ and non-descendants $\mathbf{N}_i$, the DAG asserts that X_i is independent of $\mathbf{N}_i$ given $\mathbf{P}_i$. The second component of a Bayesian network contains conditional probability distributions for each variable X_i, $Pr(X_i \mid \mathbf{P}_i)$, also known as the CPT for variable X_i. The independencies encoded by the DAG, together with these conditional distributions, define a unique distribution $Pr(X_1, \ldots, X_n)$ [Dar09, KF09, Mur12, Bar12].

Consider now the following key observation. Any set of Boolean constraints Δ implied by the network distribution $Pr(X_1, \ldots, X_n)$ must be modular in the following sense. The constraints Δ can be decomposed into sets $\Delta_1, \ldots, \Delta_n$ such that:[1]

- Δ_i mentions only variable X_i and its parents $\mathbf{P}_i$.
- Δ_i does not constrain the states of parents $\mathbf{P}_i$.

A set of constraints that satisfies the above properties will be called constraints for $X_i \mid \mathbf{P}_i$, or simply *conditional constraints* when X_i and $\mathbf{P}_i$ are clear from the context.

Here is now the key implication of the above observation.

> *If the domain under consideration admits a DAG that captures probabilistic independence, then one can encode any underlying Boolean constraints modularly, i.e., using only conditional constraints.*

The catch however is that exploiting this implication fully requires a new class of DAGs for representing independence constraints, compared to what is used in Bayesian networks. We will come back to this point later, after dwelling more on conditional constraints. In particular, we will state two properties of these constraints and provide a concrete example.

The first property of conditional constraints is this: for every parent instantiation $\mathbf{p}_i$, there is at least one state x_i that is compatible with $\mathbf{p}_i$ given Δ_i. That is, while Δ_i may eliminate some states of variable X_i under instantiation $\mathbf{p}_i$, it will never eliminate them all. We will use this property later.

The second property is that conditional constraints can always be expressed as follows. Let $\alpha_1, \ldots, \alpha_m$ be a partition of the states for parents $\mathbf{P}_i$, and let β_j be a non-empty set of states for X_i, $j = 1, \ldots, m$. Any conditional constraints for $X_i \mid \mathbf{P}_i$ can be expressed in the form "if the state of parents $\mathbf{P}_i$ is in α_j, then the state of X_i is in β_j." The converse is also true: any set of constraints of the previous form must be conditional (i.e., cannot eliminate any parent state, or eliminate all states of X_i under a given parent instantiation).

[1]If such a decomposition is not possible, we can establish a contradiction with some of the probabilistic independence constraints encoded by the DAG of the Bayesian network.

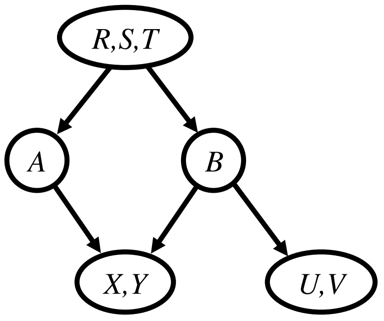


Figure 4.2: A cluster DAG.

We will now consider a concrete example of modular constraints, which are extracted from the zero parameters of a Bayesian network over three variables A, B and C. Here, variable C is a child of A and B and has the following CPT:

A	B	C	$Pr(C \mid A, B)$	A	B	C	$Pr(C \mid A, B)$
a_0	b_0	c_0	0.3	a_1	b_0	c_0	**0.0**
a_0	b_0	c_1	0.1	a_1	b_0	c_1	0.7
a_0	b_0	c_2	0.6	a_1	b_0	c_2	0.1
a_0	b_0	c_3	**0.0**	a_1	b_0	c_3	0.2
a_0	b_1	c_0	**0.0**	a_1	b_1	c_0	**0.0**
a_0	b_1	c_1	0.7	a_1	b_1	c_1	0.7
a_0	b_1	c_2	0.1	a_1	b_1	c_2	0.1
a_0	b_1	c_3	0.2	a_1	b_1	c_3	0.2

Variables A and B are binary in this case, with variable C having four states. Moreover, this CPT encodes the following conditional constraints on variable C given A and B:

– if the parents satisfy $a_1 \vee b_1$, then C satisfies $c_1 \vee c_2 \vee c_3$.

– if the parents satisfy $a_0 \wedge b_0$, then C satisfies $c_0 \vee c_1 \vee c_2$.

We will next discuss why the integration of conditional Boolean constraints with independence constraints will generally requires a new class of DAGs for representing independence constraints.

As discussed earlier, a Bayesian network over variables $X_1, \ldots, X_n$ requires one to construct a DAG in which nodes correspond to variables $X_1, \ldots, X_n$. When the Boolean constraints are massive, the resulting DAG may end up being almost fully connected, making the Bayesian network unusable practically. To address this issue, we will work with DAGs in which nodes correspond to *clusters* of variables; see Figure 4.2. The independence semantics are similar to classical Bayesian networks, except that cluster DAGs can be less committing than classical DAGs. For example, the cluster DAG in Figure 4.2 says that variables $\{X, Y\}$ are independent of $\{R, S, T\}$ (and $\{U, V\}$) given $\{A, B\}$. Cluster DAGs, however, are silent on the independence relationships between variables in the same cluster. As we shall see in the following section, the relationships among variables in the same cluster, and between a cluster and its parent clusters, will be handled by the newly introduced conditional PSDDs.

We close this section by pointing out that in this chapter, variables in a cluster DAG are assumed to be binary. Consider again the CPT above in which variable C has four states. By replacing this variable with two binary variables, X and Y, we get the following CPT:

A	B	X	Y	$Pr(X, Y \mid A, B)$
a_0	b_0	x_0	y_0	0.3
a_0	b_0	x_0	y_1	0.1
a_0	b_0	x_1	y_0	0.6
a_0	b_0	x_1	y_1	0.0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Here, state c_0 is encoded by state x_0, y_0, state c_1 is encoded by x_0, y_1 and so on, leading to the conditional constraints:

- if $a_1 \vee b_1$, then $x_1 \vee y_1$.
- if $a_0 \wedge b_0$, then $x_0 \vee y_0$.

We will refer to this example in the following section.

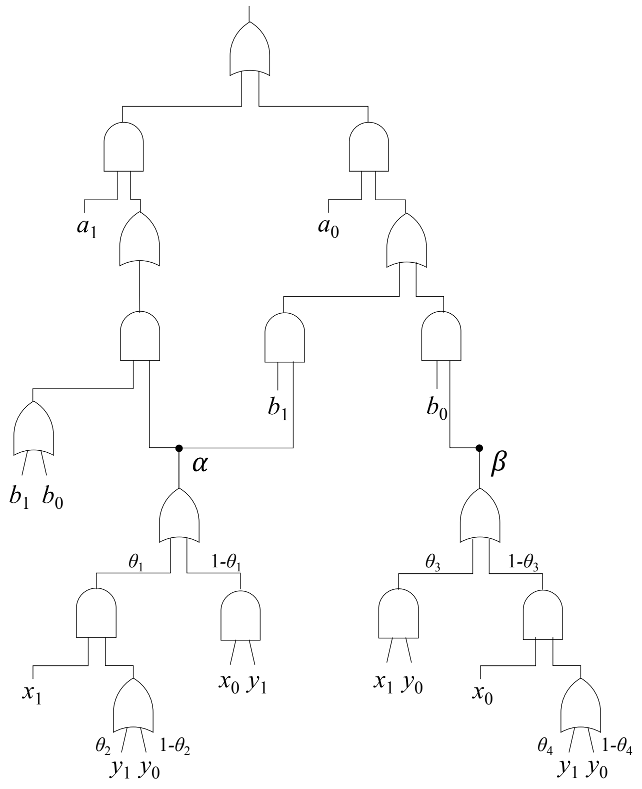


Figure 4.3: A conditional PSDD (vtree on left of Figure 4.5).

4.4 Conditional PSDDs

We will now introduce the *conditional PSDD* for representing a family of distributions that are conditioned on the same set of variables. That is, a conditional PSDD will play the role of a CPT in a Bayesian network. More precisely, a conditional PSDD will quantify the relationship between a cluster and its parents in a cluster DAG, leading to what we shall call a *structured Bayesian network.* Not only will this allow us to integrate Boolean and independence constraints into the learning process, but it will sometimes allow us to scale to networks whose nodes have exponentially many states (we will show how conditional PSDDs can be learned efficiently from complete data in the following section).

The first step in obtaining a conditional PSDD is to compile conditional constraints into an SDD. Compiling the constraints from the previous section leads to the SDD in Figure 4.3. We will next discuss two key properties of this SDD.

The first property concerns the SDDs labeled α and β. SDD α represents the Boolean

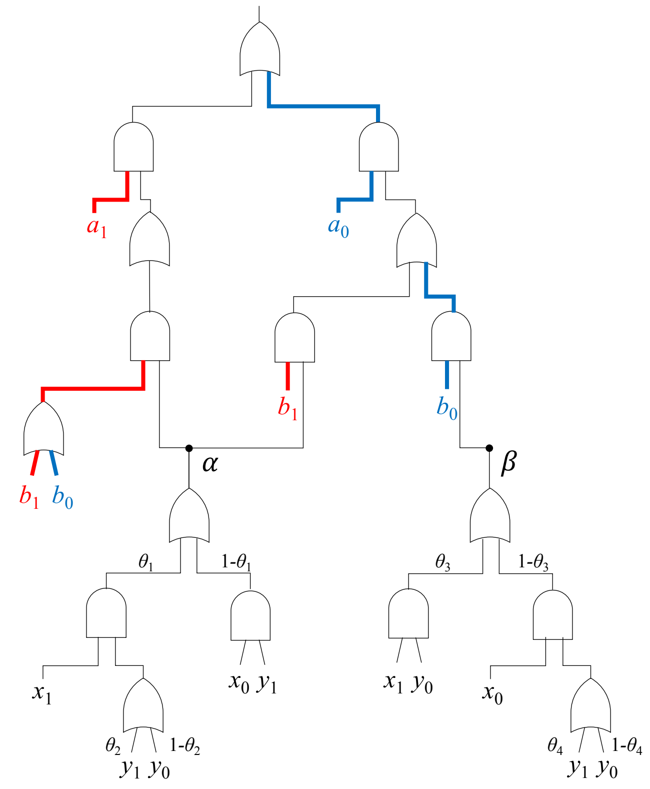


Figure 4.4: A partial evaluation of the conditional PSDD of Figure 4.3, under the input $A{=}a_1, B{=}b_1$. Wires are colored red if they are fixed to high, and colored blue if they are fixed to low. The states of uncolored wires depend on the states (values) of the inputs X and Y. In this example, the output of the circuit is the same as the value of α.

expression $x_1 \vee y_1$, which captures the states of X, Y under $a_1 \vee b_1$. SDD β represents the Boolean expression $x_0 \vee y_0$, which captures the states of X, Y under $a_0 \wedge b_0$. Each of these SDDs can be parameterized into a PSDD to yield a distribution over the corresponding states of variables X, Y; see Figure 4.3.

The second property of the SDD in Figure 4.3 is that the output of the circuit under any input that satisfies $a_1 \vee b_1$ will be the state of α; see Figure 4.4. Similarly, the output of the circuit under any input that satisfies $a_0 \wedge b_0$ will be the state of β. That is, depending on the values of parents A and B, the circuit selects either PSDD α or PSDD β. This is why the circuit in Figure 4.3 is called a *conditional PSDD:* it represents a set of PSDDs for variables X, Y, each conditioned on some state of parents A and B. According to this conditional PSDD, the distribution over variables X, Y is independent of the specific state of parents A

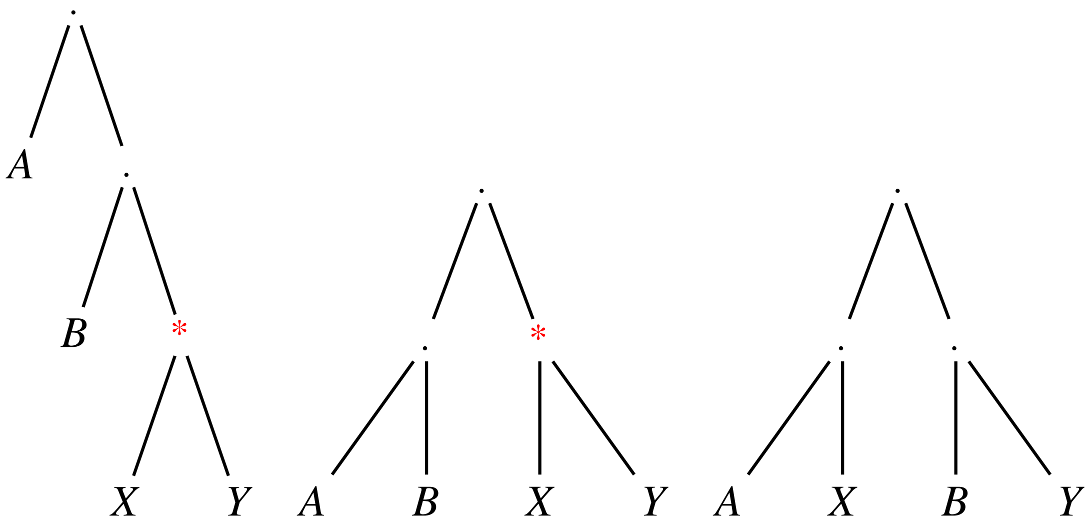


Figure 4.5: Two vtrees for $\mathbf{X} \mid \mathbf{P}$ with $\mathbf{X} = \{X, Y\}$ and $\mathbf{P} = \{A, B\}$ (left and center) and a vtree that is not for $\mathbf{X} \mid \mathbf{P}$ (right). The $\mathbf{X}$-nodes of the first two vtrees are starred.

and B, once we know that these parents satisfy $a_1 \vee b_1$. This corresponds to *context-specific independence* [BFG96], as it says that X, Y is independent of A given b_1, and independent of B given a_1. That is, the independence is conditioned on a variable taking a specific value (X, Y are neither independent of A given B nor independent of B given A). Decision trees have also been used in a similar context by [FG98], but these trees are representationally less compact than conditional PSDDs. The latter are based on SDDs, which are decision graphs (not trees) that branch on sentences instead of variables, leading to exponentially smaller representations [MT98, XCD12, Bov16].

The above properties are due to choosing a specific vtree when constructing an SDD and because the SDD was compiled from modular constraints. We formalize these next.

Definition 2 (Conditional Vtrees)**.** *Let v be a vtree for variables $\boldsymbol{X} \cup \boldsymbol{P}$ which has a node u that contains precisely the variables $\boldsymbol{X}$. If node u can be reached from node v by only following right children, then v is said to be a vtree for $\boldsymbol{X} \mid \boldsymbol{P}$ and u is said to be its $\boldsymbol{X}$-node.*

Conditional vtrees were introduced in [OCD16] for a different purpose, under the name of $\mathbf{P}$-constrained vtrees. Figure 4.5 depicts some examples of conditional vtrees for $\mathbf{X} = \{X, Y\}$ and $\mathbf{P} = \{A, B\}$. The figure also marks the $\mathbf{X}$-nodes of these vtrees.

We are now ready to formally define conditional PSDDs and their underlying SDDs.

Definition 3 (Conditional and Modular SDDs)**.** *An SDD circuit for $\boldsymbol{X} \mid \boldsymbol{P}$ is one that conforms to a vtree for $\boldsymbol{X} \mid \boldsymbol{P}$. The circuit is also modular for $\boldsymbol{X} \mid \boldsymbol{P}$ if it evaluates to* 1 *under each instantiation $\boldsymbol{p}$ and some instantiation $\boldsymbol{x}$.*

Modularity can be described in two equivalent ways: (1) the SDD does not constrain the states of parents $\mathbf{P}$, or (2) variables $\mathbf{X}$ have at least one possible state given any instantiation of parents $\mathbf{P}$. If modularity is violated by some parent instantiation $\mathbf{p}$, then we cannot represent the conditional distribution $Pr(\mathbf{X} \mid \mathbf{p})$ for that instantiation.

Definition 4 (Conditional PSDDs)**.** *An or-gate that feeds only from variables in $\boldsymbol{X}$ will be called an $\boldsymbol{X}$-gate. A PSDD for $\boldsymbol{X} \mid \boldsymbol{P}$ is a modular SDD for $\boldsymbol{X} \mid \boldsymbol{P}$ in which every $\boldsymbol{X}$-gate is parameterized.*

With $\mathbf{X} = \{X, Y\}$ and $\mathbf{P} = \{A, B\}$, we have four $\mathbf{X}$-gates in Figure 4.3. The following theorem shows that a conditional PSDD is guaranteed to contain a PSDD for each conditional distribution $Pr(\mathbf{X} \mid \mathbf{p})$.

Theorem 3. *Consider a modular SDD γ for $\boldsymbol{X} \mid \boldsymbol{P}$ and let u be the $\boldsymbol{X}$-node of its vtree. For each parent instantiation $\boldsymbol{p}$, there is a unique $\boldsymbol{X}$-gate g that (1) conforms to u and (2) has the same value as SDD γ under every circuit input $\boldsymbol{p}, \boldsymbol{x}$.*

The or-gate g will then be the root of an SDD that captures the possible states of variables $\mathbf{X}$ under parent instantiation $\mathbf{p}$. Moreover, the parameterization of this SDD leads to a PSDD for the conditional distribution $Pr(\mathbf{X} \mid \mathbf{p})$. It is critical to observe here that multiple parent instantiations $\mathbf{p}$ may map to the same or-gate g. That is, the number of PSDDs in a conditional PSDD is not necessarily exponential in the number of parents $\mathbf{P}$. More generally, the size of a conditional PSDD is neither necessarily exponential in the number of variables $\mathbf{X}$ nor their parents $\mathbf{P}$. This is the reason why structured Bayesian networks—which are cluster DAGs that are quantified by conditional PSDDs—can be viewed as Bayesian networks in which nodes can have an exponential number of states. We will later discuss a real-world application of structured Bayesian networks at length.

4.5 Learning Conditional PSDDs

Learning a conditional PSDD is akin to learning the CPT of a Bayesian network in that we are learning a set of distributions for variables $\mathbf{X}$ given their parents $\mathbf{P}$. As we shall see next, this turns out to be easy under complete data as it amounts to a process of counting examples in the dataset, which can be described using closed-form equations.

The one difference with learning CPTs is that we are now learning from constraints as well, which requires compiling the given constraints into an SDD circuit that conforms to a vtree for $\mathbf{X} \mid \mathbf{P}$. There are usually many such vtrees for a given $\mathbf{X}$ and $\mathbf{P}$ so the learning process tries to choose one that minimizes the SDD size (there is a unique compressed SDD once the vtree is fixed). A method that searches for conditional vtrees has already been proposed in [OCD16], which we adapted. [2]

Once the vtree and the SDD are fixed, the parameters of the PSDDs (embedded in our conditional PSDD) can be learned in closed form when the data is complete. This can be done using a method similar to the one proposed in [KVC14] for learning the parameters of a classical PSDD. In particular, each parameter is simply estimated by counting the number of examples in the dataset that "touches" that parameter in the PSDD. We can visualize this process using the conditional PSDD of Figure 4.6, for which the underlying SDD was evaluated under the example $A = a_1, B = b_1, X = x_1, Y = y_0$. The parameters touched by this example are the ones encountered by starting at the output or-gate, and then descending down to every gate and circuit input that evaluates to 1. According to this definition, parameters θ_1 and $1 - \theta_2$ are touched in Figure 4.6.

Suppose now that $\mathcal{D}\#(\theta_i)$ and $\mathcal{D}\#(1 - \theta_i)$ denote the number of examples in dataset $\mathcal{D}$ that touch each corresponding parameter. The (maximum-likelihood) parameters are then

[2] In a normalized and compressed SDD, two nodes cannot represent the same Boolean function if they conform to the same vtree node [Dar11]. Hence, each PSDD node conforming to the $\mathbf{X}$-node of the vtree will have a unique space over the variables $\mathbf{X}$, and hence a unique distribution for that space. This can be relaxed by either using auxiliary variables to distinguish distributions over the same space or use an SDD that is uncompressed as the underlying structure.

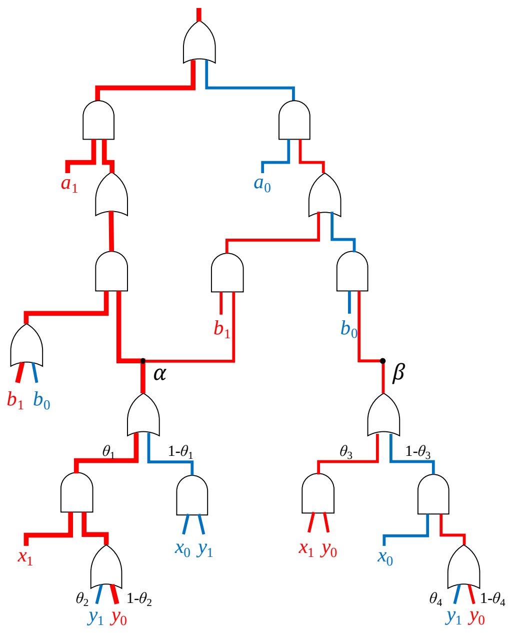


Figure 4.6: Evaluating the SDD of a conditional PSDD under the circuit input $A=a_1$, $B=b_1$, $X=x_1$, $Y=y_0$ (which can be viewed as an example in the dataset). Each wire is colored by its state (red for high and blue for low). The wires in bold are visited when descending from the root gate down to every gate and circuit input that evaluates to 1.

given by[3]

$$\theta_i^{\text{ml}} = \frac{\mathcal{D}\#(\theta_i)}{\mathcal{D}\#(\theta_i) + \mathcal{D}\#(1-\theta_i)}.$$

Again, this parameter estimation algorithm is analogous to the one given by [KVC14] for classical PSDDs, except that we do not estimate parameters for gates that are not **X**-gates. Moreover, the above parameter estimates are the maximum likelihood estimates for the structured Bayesian network that is quantified using conditional PSDDs.

4.6 Experimental Results

We now empirically evaluate our proposed approach using conditional PSDDs by contrasting it to the classical way in which PSDDs are used. In particular, given data and a cluster DAG

[3]This equation is based on or-gates with two inputs. However, it can be generalized easily to or-gates with an arbitrary number of inputs, since all underlying concepts still apply to the general case.

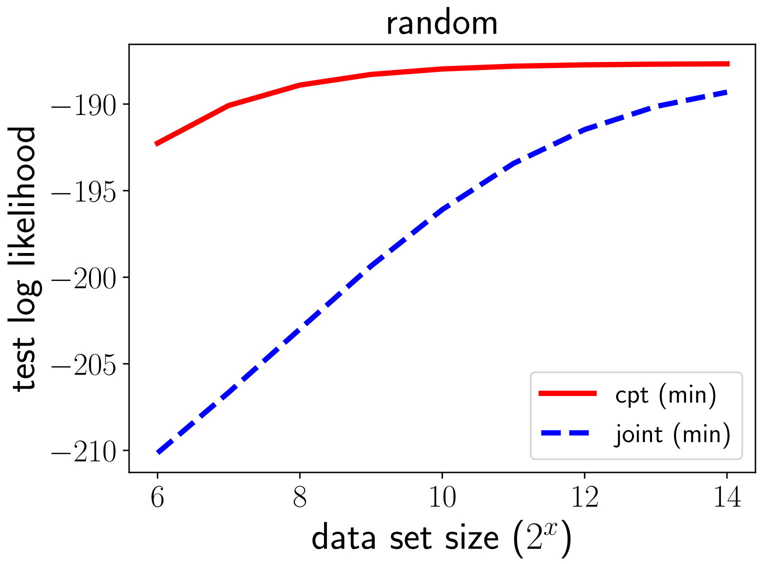


Figure 4.7: Conditional vs. joint PSDDs on random networks.

with corresponding modular constraints, we learn a model in two ways. First, we compile the constraints into an SDD that we then parameterize using the data. This is the classical method which we shall refer to as “joint PSDD.” Second, we compile each set of conditional constraints into a conditional PSDD and learn its parameters from the data. This is the proposed method which we refer to as “conditional PSDD.” We finally compare the quality of the two models learned. In each case, we also try to minimize the size of compiled SDD as is classically done.

The used data and constraints are obtained by randomly generating Bayesian network structures over 100 variables $X_1, \ldots, X_{100}$. In particular, the structure of the network is obtained by randomly selecting k parents for each variable X_i from the last s variables that precede X_i, i.e., from $X_{i-s}, \ldots, X_{i-1}$ (if they are available). For our experiments, we assumed $k = s = 4$. Each variable X_i was assumed to have $2^4 = 16$ states (hence, we used 4 SDD variables to represent that space, leading to clusters of size 4). The modular constraints were generated as follows. For each variable X_i, we divide the space of parent instantiations by asserting a random parity constraint of length $\frac{k}{2}$ where k is the number of SDD variables used to represent the parents. We then divide the space recursively, until we have a partition of size 8. For each member of the partition, we constrain variable X_i using a parity constraint of length $\frac{k}{2}$ where k is now the number of SDD variables used to represent the child (i.e., $k = 4$). Note that a parity constraints eliminates half the values of variable X_i.

We simulate datasets from this Bayesian network using forward sampling. That is, we traverse the network in topological order and draw a sample from each node given the sample of the parents. Each dataset is used to learn the conditional PSDDs of the cluster DAG, as well as the corresponding joint PSDD.

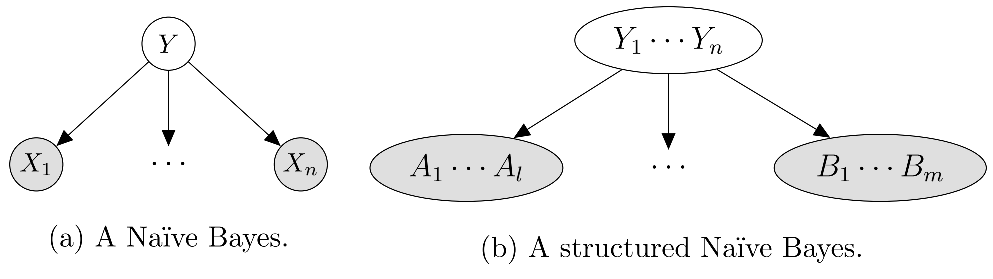
Figure 4.7 highlights the results. On the x-axis, we increase the size of the training set used. On the y-axis, we evaluate the test-set log likelihood (larger is better). We simulated 10 random networks, and for each network we simulated 7 different train/test pairs; hence, each point represents an average over 70 learning problems. For each learning problem, we increased the size of training sets from 2^6 to 2^{14}, and used an independent test set of size 2^{12}. We observe, in Figure 4.7, that the conditional PSDD obtains much better likelihoods than the joint PSDD, especially with smaller datasets. On average, the joint PSDDs had 45,618 free parameters, while the conditional PSDDs had on average 648 free parameters in total. Hence, with smaller datasets, we expect the conditional PSDDs to be more robust to overfitting.

4.7 Structured Naïve Bayes

We now discuss an application of SBNs. In particular, SBNs can be used on prediction tasks that involve combinatorial objects. We show that SBNs subsume previous methods that has tackled this task [CTD16]. In addition, SBNs provide additional capabilities to handle challenges that cannot be addressed by the previous method.

Let's first review a traditional Naïve Bayes, which is a popular Bayesian network structure that is used for the classification task. Its graphical structure is shown in Figure 4.8a. It has a single root node representing the label that we are predicting. Each leaf variable X_i corresponds to a feature, and it has the label variable as its only parent. To predict a label, one can compare between two probabilities, $Pr(\mathbf{x}, y)$ and $Pr(\mathbf{x}, \bar{y})$, and select the label configuration that results in a higher probability. A Naïve Bayes can be used to predict a simple object using a set of simple features, where each is represented as a variable in the DAG.

Its structured version, a structured Naïve Bayes, replaces each variable in a Naïve Bayes with a cluster of variables, as shown in Figure 4.8b. Each cluster models a combinatorial object tractability using a (conditional) PSDD. As a result, a structured Naïve Bayes can be used for classification tasks that cannot be modeled using a traditional Naïve Bayes; each of the features and the label can be combinatorial objects.



(a) A Naïve Bayes.

(b) A structured Naïve Bayes.

Figure 4.8: On the left, it shows a Naïve Bayes that is used for binary classification. On the right, it shows a structured Naïve Bayes that can be used for both binary and structured classification.

When the label is binary and the feature is complex, the structured Naïve Bayes corresponds to a model that has been proposed in [CTD16]. For example, the author predicts the player based on the tic-tac-toe game traces it has played. In this case, the game traces is a complex feature. [CTD16] uses a graphical assumption as shown in Figure 4.8b. The root contains a single binary variable representing the type of the player, and each leaf contains a set of variables where together represent a game trajectory that the player has played. It first uses two numbers, $Pr(\text{Player} = \text{Beginner})$ and $Pr(\text{Player} = \text{Advanced})$, to parameterize a prior probability over the variable in the root cluster, Player. In addition, it proposes a specialized method to model the conditional probability $Pr(\text{GameTrace} \mid \text{Player})$, which has an equivalent conditional PSDD representation. The conditional probability is modeled using two PSDDs, n^+ and n^-. The node n^+ represents the joint distribution of the game trajectory played by the beginner player, and n^- represents the joint distribution of the advanced player. The representation of a conditional probability, Pr(GameTrace | Player) is a special case of conditional PSDD that is formed by a template shown in Figure 4.9.[4]

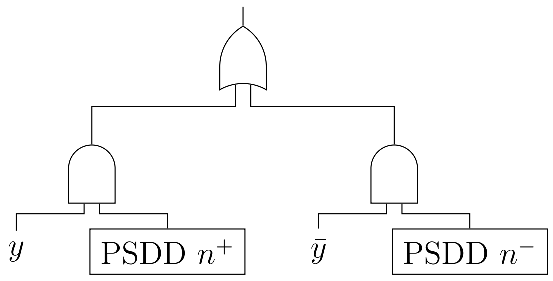


Figure 4.9: A conditional PSDD representing the CPT of a structured Naïve Bayesin [CTD16].

In addition to accommodate combinatorial features, structured Naïve Bayes can be used to predict a combinatorial object from binary features. Consider a case where we want to predict the 3 courses that are selected by a student from the 100 courses offering. In addition, we also have obtained some binary features about each student, e.g. its seniority (high/low), its GPA (below/above 2.0), etc

The problem can be modeled using a structured Naïve Bayes. The root cluster contains 100 course variables, and each variable indicates whether the student selects the corresponding course. Each leaf cluster contains a binary feature that describes the student, e.g. seniority or GPA. Next, we are going to discuss about the modeling parameters of each cluster.

To model a joint probability over a 3-choose-100 selection, methods in Chapter 3 can be used. The joint probability over the course selection can be represented as a PSDD. For each leaf cluster, a conditional PSDD is to be specified to describe the conditional probability of each binary feature given the course selection. The structure of the conditional PSDD can come from the knowledge of the context-specific independence in the problem. For example, students taking the same number of *honor* classes, which is a few selected challenging courses, have the same conditional probability of obtaining a high GPA. Based on this assumption,

[4]The underling SDD is not compressed so that we can model two distinct joint probability distributions having the same structured space, i.e. all possible game trajectories in a tic-tac-toe game.

a conditional PSDD can be crafted as in Figure 4.10 to model $Pr(X \mid \mathbf{P})$, where X is a binary variable that represents the GPA level of a student, and $\mathbf{P}$ represents the 3 selected courses by the student. Each node, n_i, in Figure 4.10 represents an SDD, and it models a logical formula that the student takes i honor classes. Whenever a student takes i honor classes, the SDD node n_i will be activated, and the corresponding terminal PSDD node, that is parameterized by θ_i, is selected to model the probability that the student has a high GPA given it takes i honor classes. In this example, the conditional PSDD is crafted from our domain knowledge. In Chapter 7, we will show that conditional PSDDs can also be learned directly from data.

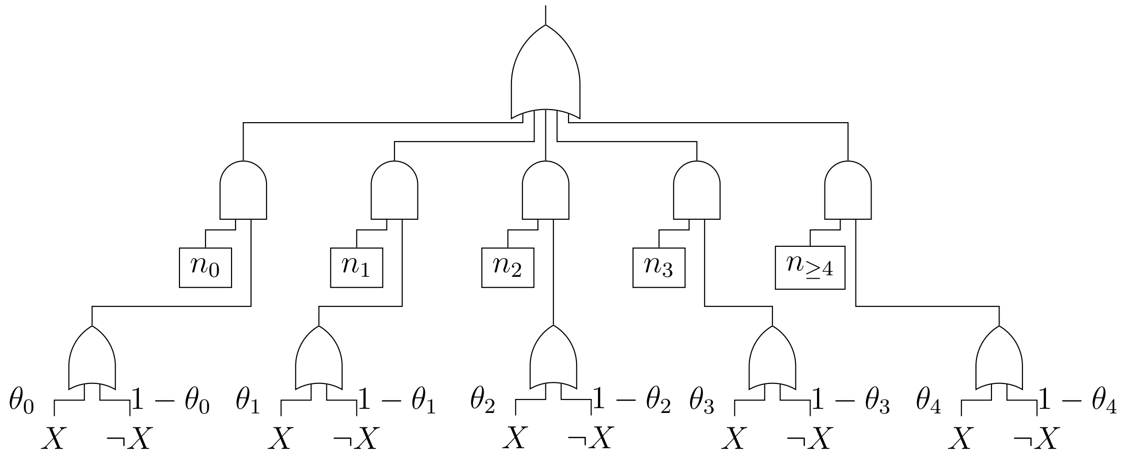


Figure 4.10: A conditional PSDD that models context specific independence.

4.8 Conclusion

We proposed conditional PSDDs and cluster DAGs, showing how they lead to a new probabilistic graphical model: The structured Bayesian network. The new model was motivated by two needs. The first is to learn probability distributions from both data and background knowledge in the form of Boolean constraints. The second is to model and exploit background knowledge that takes the form of independence constraints. The first need has been addressed previously by PSDDs, while the second has been addressed classically using probabilistic graphical models, including Bayesian networks. Structured Bayesian networks inherits the advantages of both of these representations, including closed-form parameter estimates under complete data. We presented empirical results to show the promise of structured Bayesian networks in learning better models than PSDDs, when independence information is available. We also presented a case study that tackles different prediction tasks using a sub-class of SBNs, structured Naïve Bayes. We demonstrated that a structured Naïve Bayes could model various prediction problems where combinatorial objects were involved.

4.A Proofs

To proof Theorem 3, it requires a few formal definitions. Let f be a Boolean function and $\mathbf{x}$ be an instantiation of some of its variables. Then $f|_{\mathbf{x}}$ denotes the *subfunction* obtained from f by fixing the values of variables $\mathbf{X}$ to $\mathbf{x}$. In the following proofs, we use Boolean functions and SDDs exchangeably.

A *terminal SDD* is either a variable (X), its negation ($\neg X$), false ($\bot$), or true (an or-gate with inputs X and $\neg X$).

The proof of Theorem 3 follows directly from two lemmas that we state and prove next.

Lemma 2. *Consider a modular SDD γ for $\boldsymbol{X} \mid \boldsymbol{P}$ that conforms to vtree v, and let u be the $\boldsymbol{X}$-node of vtree v. For each parent instantiation $\boldsymbol{p}$, there is an $\boldsymbol{X}$-gate g that represents the subfunction $\gamma|_{\boldsymbol{p}}$.*

Proof The proof is by induction on the level of node u in the vtree.

- Base Case: $u = v$. Then $\mathbf{P} = \emptyset$ and there is a unique (empty) instantiation $\mathbf{p}$, where $\gamma|_{\mathbf{p}} = \gamma$.

- Inductive Step $u \neq v$: Then SDD γ is a fragment; see Figure 2.2. Since SDD γ conforms to v, each prime p_i conforms to v^l and each sub s_i conforms to v^r. Since u is the $\mathbf{X}$-node, it can be reached from the root v by iteratively following right children. Hence, $\mathbf{X} \subseteq \mathsf{vars}(v^r)$ and $\mathsf{vars}(v^l) \subseteq \mathbf{P}$. Let $\mathbf{P}^l = \mathsf{vars}(v^l)$ (parent variables to the left of v) and $\mathbf{P}^r = \mathbf{P} \setminus \mathsf{vars}(v^l)$ (parent variables on the right of v). Further, let $\mathbf{p}^l$

and $\mathbf{p}^r$ denote the corresponding sub-instantiations of $\mathbf{p}$. In an SDD, the primes p_i are mutually-exclusive and exhaustive, hence $\gamma|_{\mathbf{p}^l} = s_i$ for some unique i. Moreover, since γ is modular for $\mathbf{X} \mid \mathbf{P}$, then $\gamma|_{\mathbf{p}^l}$ must be modular for $\mathbf{X} \mid \mathbf{P}^r$. Since $\gamma|_{\mathbf{p}^l} = s_i$ further conforms to v^r, then by induction there is an $\mathbf{X}$-gate g representing $s_i|_{\mathbf{p}^r}$. Gate g is also the one representing $\gamma|_{\mathbf{p}}$ since: $\gamma|_{\mathbf{p}} = (\gamma|_{\mathbf{p}^l})|_{\mathbf{p}^r} = s_i|_{\mathbf{p}^r} = g$. $\square$

Lemma 3. *Consider a modular SDD γ for $\boldsymbol{X} \mid \boldsymbol{P}$ that conforms to vtree v, and let u be the $\boldsymbol{X}$-node of vtree v. For each parent instantiation $\boldsymbol{p}$, if there is an $\boldsymbol{X}$-gate g that represents $\gamma|_{\boldsymbol{p}}$, then gate g (1) conforms to u, (2) evaluates to 1 on $\boldsymbol{x}$ iff γ evaluates to 1 on $\boldsymbol{p}, \boldsymbol{x}$, and (3) is unique.*

The proof uses a property of normalized and compressed SDDs that underlie conditional PSDDs. That is, in such SDDs, we cannot have two distinct or-gates that conform to the same vtree node yet represent the same Boolean function [Dar11].

Proof We prove each part in turn.

1. By Lemma 2, there is an $\mathbf{X}$-gate g representing $\gamma|_{\mathbf{p}}$. From the proof of Lemma 2, this $\mathbf{X}$-gate g is obtained by following a path in the circuit γ through the subs. By conformity, each of these subs comform to the corresponding right child of a vtree node. Hence, g must conform to u.

2. For a given instantiation $\mathbf{p}$, we have $\gamma(\mathbf{p}, \mathbf{x}) = \gamma|_{\mathbf{p}}(\mathbf{x})$, which is equivalent to $g(\mathbf{x})$ from the proof of Lemma 2.

3. Suppose that there exists a gate h that conforms to u and which evaluates to 1 on $\mathbf{x}$ iff γ evaluates to 1 on $\mathbf{p}, \mathbf{x}$. It follows that $h(\mathbf{x}) = g(\mathbf{x})$ for all $\mathbf{x}$. Since the SDD is normalized and compressed, we must have $h = g$. $\square$

CHAPTER 5

Inference

In this chapter, we illustrate an exact inference algorithm for SBNs. The algorithm is based on compiling the SBN to a PSDD, where both models represent the identical joint probability distribution. After the PSDD is obtained, many tractable probabilistic queries can be computed in time that is linear in the size of the PSDD, see Chapter 2. The results discussed in this chapter appeared in [SCD16] and [SGD19].

5.1 Introduction

Most inference tasks on graphical models are computationally challenging, which is also the case for SBNs. For example, to calculate the marginal probability of a variable in a regular BN is known to be PP-Complete [Coo90, Rot96]. As any regular BN is a sub-class of SBNs, computing the marginal query on an SBN is at least as hard. Due to this complexity, many works have suggested to compile the joint distribution of a graphical model to a more tractable representation [CKD13, CD07]. The compilation algorithms can have the same time complexity as the traditional junction tree algorithm in the worst case [Dar09]. After the tractable representation is obtained, most probabilistic queries can be answered in time that is linear to the size of the representation. As local structures are exploited during the compilation but not in the vanilla version of a join tree algorithm, inferencing on a tractable representation can be exponentially faster. Due to these advantages, we are interested in compiling the SBNs into a more tractable representation, i.e. PSDDs.

The chapter is organized as follows. In Section 5.2, we review different representations of probability distributions. We study two properties of the representation, which enable it to support multiple tractable probabilistic queries. In Section 5.3, we study different tractable operations that PSDD supports. In Section 5.4, we use one of the introduced tractable operation, multiple, to compile a traditional graphical model into a single PSDD. Furthermore, we show that a similar procedure can also be used to compile an SBN into a single PSDD in Section 5.5. Finally, we provide an empirical analysis in Section 5.6 and conclude in Section 5.7.

5.2 Representing Distributions Using Arithmetic Circuits

We start with the definition of factors, which include distributions as a special case.

Definition 5 (Factor)**.** *A* factor $f(\boldsymbol{X})$ *over variables* $\boldsymbol{X}$ *maps each instantiation* $\boldsymbol{x}$ *of variables* $\boldsymbol{X}$ *into a non-negative number* $f(\boldsymbol{x})$*. If* $\sum_{\boldsymbol{x}} f(\boldsymbol{x}) = 1$*, the factor represents a distribution.*

We define the value of a factor at a partial instantiation $\mathbf{y}$, where $\mathbf{Y} \subseteq \mathbf{X}$, as $f(\mathbf{y}) = \sum_{\mathbf{z}} f(\mathbf{yz})$, where $\mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$. When the factor is a distribution, $f(\mathbf{y})$ corresponds to the probability of evidence $\mathbf{y}$. We also define the MAP instantiation of a factor as $\operatorname{argmax}_{\mathbf{x}} f(\mathbf{x})$, which corresponds to the most likely instantiation when the factor is a distribution.

The classical, tabular representation of a factor $f(\mathbf{X})$ is exponential in the number of variables $\mathbf{X}$. However, one can represent such factors more compactly using arithmetic circuits.

Definition 6 (Arithmetic Circuit)**.** *An arithmetic circuit* $\mathcal{AC}(\boldsymbol{X})$ *over variables* $\boldsymbol{X}$ *is a rooted DAG whose internal nodes are labeled with* $+$ *or* $*$ *and whose leaf nodes are labeled with either indicator variables* λ_x *or non-negative parameters* θ*. The value of the circuit at instantiation* $\boldsymbol{x}$*, denoted* $\mathcal{AC}(\boldsymbol{x})$*, is obtained by assigning indicator* λ_x *the value* 1 *if* x *is compatible with instantiation* $\boldsymbol{x}$ *and* 0 *otherwise, then evaluating the circuit in the standard way. The circuit* $\mathcal{AC}(\boldsymbol{X})$ *represents factor* $f(\boldsymbol{X})$ *iff* $\mathcal{AC}(\boldsymbol{x}) = f(\boldsymbol{x})$ *for each instantiation* $\boldsymbol{x}$*.*

A *tractable* arithmetic circuit allows one to efficiently answer certain queries about the factor it represents. We next discuss two properties that lead to tractable arithmetic circuits. The first is *decomposability* [Dar01a], which was used for probabilistic reasoning [Dar03].

Definition 7 (Decomposability). *Let n be a node in an arithmetic circuit $\mathcal{AC}(\boldsymbol{X})$. The variables of n, denoted $vars(n)$, are the variables $X \in \boldsymbol{X}$ with some indicator λ_x appearing at or under node n. An arithmetic circuit is decomposable iff every pair of children c_1 and c_2 of a $*$-node satisfies $vars(c_1) \cap vars(c_2) = \emptyset$.*

The second property is *determinism* [Dar01b], which was also employed for probabilistic reasoning in [Dar03].

Definition 8 (Determinism). *An arithmetic circuit $\mathcal{AC}(\boldsymbol{X})$ is deterministic iff each $+$-node has at most one non-zero input when the circuit is evaluated under any instantiation $\boldsymbol{x}$ of the variables $\boldsymbol{X}$.*

A third property called *smoothness* is also desirable as it simplifies the statement of certain AC algorithms, but is less important for tractability as it can be enforced in poly-time [Dar01b].

Definition 9 (Smoothness). *An arithmetic circuit $\mathcal{AC}(\boldsymbol{X})$ is smooth iff it contains at least one indicator for each variable in $\boldsymbol{X}$, and for each child c of $+$-node n, we have $vars(n) = vars(c)$.*

Decomposability and determinism lead to tractability in the following sense. Let $Pr(\mathbf{X})$ be a distribution represented by a decomposable, deterministic and smooth arithmetic circuit $\mathcal{AC}(\mathbf{X})$. Then one can compute the following queries in time that is linear in the size of circuit $\mathcal{AC}(\mathbf{X})$: the probability of any partial instantiation, $Pr(\mathbf{y})$, where $\mathbf{Y} \subseteq \mathbf{X}$ [Dar03] and the most likely instantiation, $\text{argmax}_{\mathbf{x}}\, Pr(\mathbf{x})$ [CD06]. The decision problems of these queries are known to be PP-complete and NP-complete for Bayesian networks [Rot96, Shi94].

A number of methods have been proposed for compiling a Bayesian network into a decomposable, deterministic and smooth AC that represents its distribution [Dar03]. Figure 5.1 depicts such a circuit that represents the distribution of Bayesian network $A \rightarrow B$. One

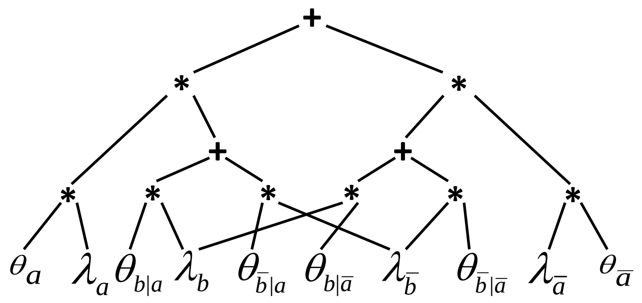


Figure 5.1: An AC for a Bayesian network $A \to B$.

method ensures that the size of the AC is proportional to the size of a jointree for the network. Another method yields circuits that can sometimes be exponentially smaller, and is implemented in the publicly available ace system [CD08]; see also [DDC08]. Additional methods are discussed in [Dar09, chapter 12].

This work is motivated by the following limitation of these tractable circuits, which may narrow their applicability in probabilistic reasoning and learning.

Definition 10 (Multiplication)**.** *The product of two arithmetic circuits $\mathcal{AC}_1(\boldsymbol{X})$ and $\mathcal{AC}_2(\boldsymbol{X})$ is an arithmetic circuit $\mathcal{AC}(\boldsymbol{X})$ such that $\mathcal{AC}(\boldsymbol{x}) = \mathcal{AC}_1(\boldsymbol{x})\mathcal{AC}_2(\boldsymbol{x})$ for every instantiation $\boldsymbol{x}$.*

Theorem 4. *Computing the product of two decomposable ACs is NP-hard if the product is also decomposable. Computing the product of two decomposable and deterministic ACs is NP-hard if the product is also decomposable and deterministic.*

We now investigate a class of tractable ACs, called the Probabilistic Sentential Decision Diagram (PSDD) [KVC14]. In particular, we show that this class of circuits admits a tractable product operation and then explore an application of this operation to exact inference in probabilistic graphical models.

Definition 11 (ACs of PSDDs)**.** *The arithmetic circuit of a PSDD is obtained as follows. Leaf nodes x and $\perp$ are converted into λ_x and 0, respectively. Each and-gate is converted into a $*$-node. Each or-node with children $c_1, \ldots, c_n$ and corresponding parameters $\alpha_1, \ldots, \alpha_n$ is converted into a $+$-node with children $\alpha_1 * c_1, \ldots, \alpha_n * c_n$.*

Theorem 5. *The arithmetic circuit of a PSDD represents the distribution induced by the PSDD. Moreover, the arithmetic circuit is decomposable and deterministic* [1]*.*

5.3 Operation on PSDDs

Factors and their operations are fundamental to probabilistic inference, whether exact or approximate [Dar09, KF09]. Consider two of the most basic operations on factors: (1) computing the product of two factors and (2) summing out a variable from a factor. With these operations, one can directly implement various inference algorithms, including variable elimination, the jointree algorithm, and message-passing algorithms such as loopy belief propagation. Typically, tabular representations (and their sparse variations) are used to represent factors and implement the above algorithms; see [LD03, SM05, CD07] for some alternatives.

More generally, factor multiplication is useful for online or incremental reasoning with probabilistic models. In some applications, we may not have access to all factors of a model beforehand, to compile as a jointree or an arithmetic circuit. For example, when learning the structure of a Markov network from data [BDC15], we may want to introduce and remove candidate factors from a model, while evaluating the changes to the log likelihood. Certain realizations of generalized belief propagation also require the multiplication of factors [YFW05, CD11]. In these realizations, one can use factor multiplication to enforce dependencies between factors that have been relaxed to make inference more tractable, albeit less accurate.

In this section, we discuss two operations of PSDDs. We first discuss multiplying two PSDDs. Then, we discuss the problem of summing out a variable from a PSDD.

Algorithm 2 Multiply(n_1, n_2, v)

```
input: PSDDs $n_1, n_2$ normalized for vtree $v$
output: PSDD $n$ and constant $\kappa$
main:
 1: $n, k \leftarrow \mathsf{cache_m}(n_1, n_2), \mathsf{cache_c}(n_1, n_2)$          ▷ check if previously computed
 2: if $n \neq$ null then return $(n, k)$                          ▷ return previously cached result
 3: else if $v$ is a leaf then $(n, \kappa) \leftarrow$ BaseCase$(n_1, n_2)$   ▷ $n_1, n_2$ are literals, $\bot$ or simple or-gates
 4: else                                                  ▷ $n_1$ and $n_2$ have the structure in Figure 2.2
 5:     $\gamma, \kappa \leftarrow \{\}, 0$                                          ▷ initialization
 6:     for all elements $(p, s, \alpha)$ of $n_1$ do                               ▷ see Figure 2.2
 7:         for all elements $(q, r, \beta)$ of $n_2$ do                            ▷ see Figure 2.2
 8:             $(m_1, k_1) \leftarrow$ Multiply$(p, q, v^l)$             ▷ recursively multiply primes $p$ and $q$
 9:             if $k_1 \neq 0$ then                            ▷ if $(m_1, k_1)$ is not a trivial factor
10:                 $(m_2, k_2) \leftarrow$ Multiply$(s, r, v^r)$         ▷ recursively multiply subs $s$ and $r$
11:                 $\eta \leftarrow k_1 \cdot k_2 \cdot \alpha \cdot \beta$            ▷ compute weight of element $(m_1, m_2)$
12:                 $\kappa \leftarrow \kappa + \eta$                        ▷ aggregate weights of elements
13:                 add $(m_1, m_2, \eta)$ to $\gamma$
14:     $\gamma \leftarrow \{(m_1, m_2, \eta/\kappa) \mid (m_1, m_2, \eta) \in \gamma\}$     ▷ normalize parameters of $\gamma$
15:     $n \leftarrow$ unique PSDD node with elements $\gamma$          ▷ cache lookup for unique nodes
16: $\mathsf{cache_m}(n_1, n_2) \leftarrow n$
17: $\mathsf{cache_c}(n_1, n_2) \leftarrow \kappa$                                ▷ store results in cache
18: return $(n, \kappa)$
```

5.3.1 Multiplying Two PSDDs

Our first observation is that the product of two distributions is generally not a distribution, but a factor. Moreover, a factor $f(\mathbf{X})$ can always be represented by a distribution $Pr(\mathbf{X})$ and a constant κ such that $f(\mathbf{x}) = \kappa \cdot Pr(\mathbf{x})$. Hence, our proposed multiplication method will output a PSDD together with a constant, as given in Algorithm 2. This algorithm uses three caches, one for storing constants ($\mathsf{cache_c}$), another for storing circuits ($\mathsf{cache_m}$), and a third used to implement Line 15 [2]. This line ensures that the PSDD has no duplicate structures of the form given in Figure 2.2. The details of function BaseCase() on Line 3 are omitted for space limitations.

The following theorem establishes the soundness and complexity of the given algorithm.

Theorem 6. *Algorithm 2 outputs a PSDD n normalized for vtree v. Moreover, if $Pr_1(\boldsymbol{X})$ and $Pr_2(\boldsymbol{X})$ are the distributions of input PSDDs n_1 and n_2, and $Pr(\boldsymbol{X})$ is the distribution of output PSDD n, then $Pr_1(\boldsymbol{x})Pr_2(\boldsymbol{x}) = \kappa \cdot Pr(\boldsymbol{x})$ for every instantiation $\boldsymbol{x}$. Finally, Algorithm 2 takes time $O(s_1 s_2)$, where s_1 and s_2 are the sizes of input PSDDs.*

We will later discuss an application of PSDD multiplication to probabilistic inference, in which we cascade these multiplication operations. In particular, we end up multiplying two factors f_1 and f_2, represented by PSDDs n_1 and n_2 and the corresponding constants κ_1 and κ_2. We use Algorithm 2 for this purpose, multiplying PSDDs n_1 and n_2 (distributions), to yield a PSDD n (distribution) and a constant κ. The factor $f_1 f_2$ will then correspond to PSDD n and constant $\kappa \cdot \kappa_1 \cdot \kappa_2$.

Another observation is that Algorithm 2 assumes that the input PSDDs are over the same vtree and, hence, same set of variables. A more detailed version of this algorithm can multiply two PSDDs over different sets of variables as long as the PSDDs have *compatible* vtrees. We omit this version here to simplify the presentation, but mention that it has the same complexity as Algorithm 2.

[1] *The arithmetic circuit also satisfies a minor weakening of smoothness with the same effect as smoothness.*

[2] The cache key of a PSDD node in Figure 2.2 is based on the (unique) ID's of nodes p_i/s_i and parameters α_i.

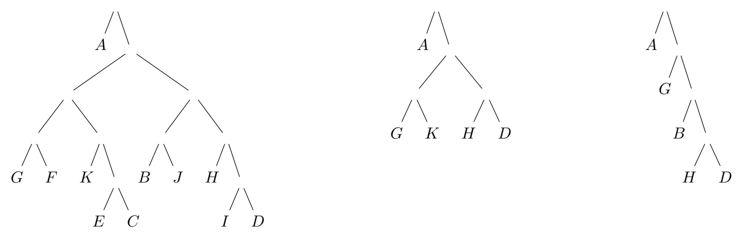


Figure 5.2: A vtree and two of its projections.

Two vtrees over variables $\mathbf{X}$ and $\mathbf{Y}$ are compatible iff they can be obtained by projecting some vtree on variables $\mathbf{X}$ and $\mathbf{Y}$, respectively.

Definition 12 (Vtree Projection)**.** *Let v be a vtree over variables $\boldsymbol{Z}$. The projection of v on variables $\boldsymbol{X} \subseteq \boldsymbol{Z}$ is obtained as follows. Successively remove every maximal subtree v' whose variables are outside $\boldsymbol{X}$, while replacing the parent of v' with its sibling.*

Figure 5.2 depicts a vtree and two of its projections. When compiling a probabilistic graphical model into a PSDD, we first construct a vtree v over all variables in the model. We then compile each factor $f(\mathbf{X})$ into a PSDD, using the projection of v on variables $\mathbf{X}$. We finally multiply the PSDDs of these factors.

5.3.2 Summing-Out a Variable in a PSDD

We now discuss the summing out of variables from distributions represented by arithmetic circuits.

Definition 13 (Sum Out)**.** *Summing-out a variable $X \in \boldsymbol{X}$ from factor $f(\boldsymbol{X})$ results in another factor over variables $\boldsymbol{Y} = \boldsymbol{X} \setminus \{X\}$, denoted by $\sum_X f$ and defined as:*

$$\left(\sum_X f\right)(\boldsymbol{y}) \overset{def}{=} \sum_x f(x, \boldsymbol{y}).$$

When the factor is a distribution (i.e., normalized), the sum out operation corresponds to marginalization. Together with multiplication, summing out provides a direct implementation of algorithms such as variable elimination and those based on message passing.

Just like multiplication, summing out is also intractable for a common class of arithmetic circuits.

Theorem 7. *The sum-out operation on decomposable and deterministic ACs is NP-hard, assuming the output is also decomposable and deterministic.*

This theorem does not preclude the possibility that the resulting AC is of polynomial size with respect to the size of the input AC—it just says that the computation is intractable. Summing out is also intractable on PSDDs, but the result is stronger here as the size of the output can be exponential.

Theorem 8. *There exists a class of factors $f(\boldsymbol{X})$ and variable $X \in \boldsymbol{X}$, such that $n = |\boldsymbol{X}|$ can be arbitrarily large, $f(\boldsymbol{X})$ has a PSDD whose size is linear in n, while the PSDD of $\sum_X f$ has size exponential in n for every vtree.*

Only the multiplication operation is needed to compile probabilistic graphical models into arithmetic circuits. Even for inference algorithms that require summing out variables, such as variable elimination, summing out can still be useful, even if intractable, since the size of resulting arithmetic circuit will not be larger than a tabular representation.

5.4 Compiling Probabilistic Graphical Models into PSDDs

Even though PSDDs form a strict subclass of decomposable and deterministic ACs (and satisfy stronger properties), one can still provide the following classical guarantee on PSDD size.

Theorem 9. *The connectivity graph of factors $f_1(\boldsymbol{X}_1), \ldots, f_n(\boldsymbol{X}_n)$ has nodes corresponding to variables $\boldsymbol{X}_1 \cup \ldots \cup \boldsymbol{X}_n$ and an edge between two variables iff they appear in the same factor. There is a PSDD for the product $f_1 \ldots f_n$ whose size is $O(m \cdot \exp(w))$, where m is the number of variables and w is its treewidth.*

This theorem provides an upper bound on the size of PSDD compilations for both Bayesian and Markov networks. An analogous guarantee is available for SDD circuits of propositional models, using a special type of vtree known as a *decision vtree* [OD14]. We next discuss our experiments, which focused on the compilation of Markov networks using decision vtrees.

To compile a Markov network, we first construct a decision vtree using a known technique.[3] For each factor of the network, we project the vtree on the factor variables, and then compile the factor into a PSDD. This can be done in time linear in the factor size, but we omit the details here. We finally multiply the obtained PSDDs. The order of multiplication is important to the overall efficiency of the compilation approach. The order we used is as follows. We assign each PSDD to the lowest vtree node containing the PSDD variables, and then multiply PSDDs in the order that we encounter them as we traverse the vtree bottom-up (this is analogous to compiling CNFs in [CKD13]).

Table 5.1 summarizes our results. We compiled Markov networks into three types of arithmetic circuits. The first compilation (AC1) is to decomposable and deterministic ACs using ace [CD08].[4] The second compilation (AC2) is also to decomposable and deterministic ACs, but using the approach proposed in [CKD13]. The third compilation is to PSDDs as discussed above. The first two approaches are based on reducing the inference problem into a weighted model counting problem. In particular, these approaches encode the network using Boolean expressions, which are compiled to logical representations (d-DNNF or SDD), from which an arithmetic circuit is induced. The systems underlying these approaches are quite complex and are the result of many years of engineering. In contrast, the proposed compilation to PSDDs does not rely on an intermediate representation or additional boxes, such as d-DNNF or SDD compilers.

The benchmarks in Table 5.1 are from the UAI-14 Inference Competition.[5] We selected all networks over binary variables in the MAR track, and report a network only if at least one approach successfully compiled it (given time and space limits of 30 minutes and 16GB).

[3]We used the minic2d package which is available at `reasoning.cs.ucla.edu/minic2d/`.

[4]The ace system is publicly available at `http://reasoning.cs.ucla.edu/ace/`.

[5]`http://www.hlt.utdallas.edu/~vgogate/uai14-competition/index.html`

Table 5.1: AC compilation size (number of edges) and time (in seconds)

	compilation size			compilation time		
network	AC1	AC2	psdd	AC1	AC2	psdd
Alchemy_11	12,705,213	-	13,715,906	130.83	-	300.80
Grids_11	81,074,816	-	-	271.97	-	-
Grids_12	232,496	457,529	201,250	0.93	1.12	1.68
Grids_13	81,090,432	-	-	273.88	-	-
Grids_14	83,186,560	-	-	279.12	-	-
Segmentation_11	20,895,884	41,603,129	30,951,708	72.39	204.54	223.60
Segmentation_12	15,840,404	41,005,721	34,368,060	51.27	209.03	283.79
Segmentation_13	33,746,511	78,028,443	33,726,812	117.46	388.97	255.29
Segmentation_14	16,965,928	48,333,027	46,363,820	62.31	279.19	639.07
Segmentation_15	29,888,972	-	33,866,332	107.87	-	273.67
Segmentation_16	18,799,112	54,557,867	19,935,308	65.64	265.07	163.38
relational_3	-	183,064	41,070	-	1.21	10.43
relational_5	-	-	217,696	-	-	594.68
Promedus_11	67,036	174,592	30,542	6.80	1.88	2.28
Promedus_12	45,119	349,916	48,814	0.91	5.81	2.46
Promedus_13	42,065	83,701	26,100	0.80	0.23	3.94
Promedus_14	2,354,180	3,667,740	749,528	21.64	33.27	24.90
Promedus_15	14,363	31,176	9,520	0.95	0.10	1.52
Promedus_16	45,935	154,467	29,150	1.35	0.40	2.06
Promedus_17	3,336,316	9,849,598	1,549,170	68.08	48.47	50.22
Promedus_18	3,006,654	762,247	539,478	20.46	18.38	21.20
Promedus_19	796,928	1,171,288	977,510	6.80	25.01	68.62
Promedus_20	70,422	188,322	70,492	0.96	3.24	3.46
Promedus_21	17,528	31,911	10,944	0.62	0.18	1.78
Promedus_22	26,010	39,016	33,064	0.63	0.10	1.58
Promedus_23	329,669	1,473,628	317,514	3.29	17.77	10.88
Promedus_24	4,774	9,085	1,960	0.45	0.05	0.80
Promedus_25	556,179	3,614,581	407,974	7.66	32.90	6.78
Promedus_26	57,190	24,578	5,146	0.71	198.74	2.72
Promedus_27	33,611	52,698	19,434	0.73	0.55	1.16
Promedus_28	24,049	46,364	17,084	1.04	0.30	1.59
Promedus_29	10,403	20,600	4,828	0.54	0.08	1.88
Promedus_30	9,884	21,230	6,734	0.50	0.07	1.23
Promedus_31	17,977	31,754	10,842	0.57	0.12	1.96
Promedus_32	15,215	33,064	8,682	0.59	0.11	1.77
Promedus_33	10,734	18,535	4,006	0.59	0.07	1.57
Promedus_34	38,113	54,214	21,398	0.87	0.78	1.78
Promedus_35	18,765	31,792	11,120	0.68	0.13	1.79
Promedus_36	19,175	31,792	11,004	1.22	0.12	1.91
Promedus_37	77,088	144,664	79,210	1.49	3.50	6.15
Promedus_38	177,560	593,675	123,552	1.67	17.19	8.09

We report the size (the number of edges) and time spent for each compilation. First, we note that for all benchmarks that compiled to both PSDD and AC2 (based on SDDs), the PSDD size is always smaller. This can be attributed in part to the fact that reductions to weighted model counting represent parameters explicitly as variables, which are retained throughout the compilation process. In contrast, PSDD parameters are annotated on its edges. More interestingly, when we multiply two PSDD factors, the parameters of the inputs may not persist in the output PSDD. That is, the PSDD only maintains enough parameters to represent the resulting distribution, which further explains the size differences.

In the `Promedus` benchmarks, we also see that in all but 5 cases, the compiled PSDD is smaller than AC1. However, several `Grids` benchmarks were compilable to AC1, but failed to compile to AC2 or PSDD, given the time and space limits. On the other hand, we were able to compile some of the `relational` benchmarks to PSDD, which did not compile to AC1 and compiled partially to AC2.

5.5 Inference in SBNs by Compilation to PSDDs

In this section, we propose the first exact inference algorithm for structured Bayesian networks (SBNs). Our approach is based on the method for compiling graphical models into PSDDs, which we summarize below:

1. pick a decision vtree v for the given Bayesian network N, using min-fill for example;
2. compile each CPT of network N into a PSDD using the vtree v projected onto the CPT's variables;
3. using the PSDD multiply operator, multiply all CPTs.

The result is a single PSDD representing the joint distribution of the given BN; we shall refer to this PSDD as the *joint PSDD*. Once we obtain the joint PSDD, we can perform exact inference efficiently: we can compute marginals or MPEs, for example, in time linear in the size of the PSDD [KVC14]. Moreover, one can bound the size of the joint PSDD of a BN by its treewidth, by using an appropriate vtree.

To perform exact inference in SBNs, we also perform three similar steps, with Step (3) being the same for SBNs as it is for BNs: each conditional PSDD can be treated as a PSDD, which we can multiply together. Step (1) is also similar for SBNs, but we will need another method for picking a special type of vtrees for SBNs, which we discuss later. Step (2) is the main difference. In order to multiply two PSDDs together using the PSDD multiply operator, the vtrees of the two PSDDs must be compatible, i.e., they are the projections of the same vtree. This is easy to ensure in a BN, as we simply convert each CPT to a PSDD using the vtree from Step (1). This is not easy to ensure in an SBN, since their conditional distributions must be specified as a conditional PSDD already, whose conditional vtrees may not be compatible.[6] Hence, for SBNs, in Step (1), we need to make sure we pick the right

[6]In a classical BN, a tabular CPT is learned from data, which is then easy to convert into a PSDD for the purposes of inference. In an SBN, conditional PSDDs represent complex conditional distributions that would otherwise have intractable representations as tables; hence, a conditional PSDD must typically be learned from data directly. In addition, when learning conditional PSDDs, we will want to learn their conditional vtrees independently (to provide the best fit to the data).

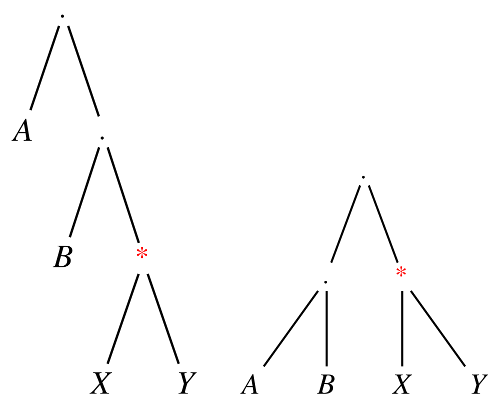


Figure 5.3: Conditional Vtrees

vtree that will allow us to, in Step (2), enforce compatibility. We discuss how to do this next.

5.5.1 Vtrees

In the compilation algorithm that we propose next, the *conditional vtree* and the *decision vtree* will be central. First, for an internal vtree node v, we refer to v^l and v^r as the left and right children of v. We call an internal vtree node v a *Shannon node* iff its left child is a leaf node. Consider the following definition of a decision vtree, originally proposed by [OCD16].

Definition 14 (Decision Vtree)**.** *A family $\boldsymbol{X} \mid \boldsymbol{P}$ is compatible with an internal vtree node v iff the family has some variables mentioned in v^l and some variables mentioned in v^r. A vtree for an SBN N is said to be a* <u>*decision vtree*</u> *for N iff every family in N is compatible with only Shannon nodes.*

Next, we consider conditional vtrees. Any conditional PSDD must conform to a conditional vtree.

Definition 15 (Conditional Vtrees)**.** *Let v be a vtree for variables $\boldsymbol{X} \cup \boldsymbol{P}$ which has a node u that contains precisely the variables $\boldsymbol{X}$. If node u can be reached from node v by only following right children, then v is said to be a* <u>*conditional vtree*</u> *for $\boldsymbol{X} \mid \boldsymbol{P}$ and u is said to be its $\boldsymbol{X}$-node.*

Figure 5.3 depicts two examples of conditional vtrees for $\mathbf{X} = \{X, Y\}$ and $\mathbf{P} = \{A, B\}$.

Algorithm 3 ConstructDecisionCtree(cluster DAG $\mathcal{B}$, topological ordering π)

```
 1: if $\mathcal{B}$ is a single cluster $\mathbf{X}$ then return a leaf ctree for cluster $\mathbf{X}$
 2: else if $\mathcal{B}$ is disconnected then
 3:     $\mathcal{B}_1, \mathcal{B}_2 \leftarrow$ a disconnected partition of $\mathcal{B}$
 4:     $\pi_1, \pi_2 \leftarrow$ sub-orders over clusters in $\mathcal{B}_1, \mathcal{B}_2$ of the total ordering $\pi$
 5: else
 6:     $\mathbf{X} \leftarrow$ first cluster of ordering $\pi$
 7:     $\mathcal{B}_1, \mathcal{B}_2 \leftarrow$ root cluster $\mathbf{X}$, and cluster DAG $\mathcal{B}$ with root cluster $\mathbf{X}$ removed
 8:     $\pi_1, \pi_2 \leftarrow$ ordering $\langle \mathbf{X} \rangle$, and ordering $\pi$ with the first element $\mathbf{X}$ removed
 9: $v^l \leftarrow$ ConstructDecisionCtree($\mathcal{B}_1, \pi_1$).
10: $v^r \leftarrow$ ConstructDecisionCtree($\mathcal{B}_2, \pi_2$).
11: return a ctree $v$ with with left and right children $v^l$ and $v^r$
```

The **X**-nodes are starred. The vtree under the **X**-node determines the circuit structure of the probabilistic (PSDD) component of a conditional PSDD. In turn, the vtree outside of the **X**-node determines the circuit structure of the logical (SDD) component.

Finally, we say that two vtrees, one over variables $\mathbf{X}$ and the other over variables $\mathbf{Y}$, are compatible iff they can be obtained by projecting some other common vtree on variables $\mathbf{X}$ and $\mathbf{Y}$, respectively.

Definition 16 (Vtree Projection)**.** *Let v be a vtree over variables $\boldsymbol{Z}$. The projection of v on variables $\boldsymbol{X} \subseteq \boldsymbol{Z}$ is obtained as follows. Successively remove every maximal subtree v_0 whose variables are outside $\boldsymbol{X}$, while replacing the parent of v_0 with its sibling.*

5.5.2 Finding a Global Vtree

In this section, we consider analogues of vtrees (and their variations) for SBNs, where leaves are labeled by clusters of the SBN, rather than by variables; we refer to these cluster trees as *ctrees*. A ctree can be viewed as a restricted type of vtree: a ctree is a vtree where the variables $\mathbf{X}$ of a cluster appear in the same sub-vtree, but where we represent this sub-vtree with a single leaf ctree node. Note, however, that we shall leave the sub-vtree over variables

X implicit for now, and first show how to pick a ctree.

Our goal is to identify a ctree for a given SBN that will accommodate inference by PSDD multiplication. The first requirement is that the ctrees of all conditional PSDDs must be compatible with some common joint ctree. The second requirement is that these ctrees must also be conditional ctrees. The first requirement can be enforced by insisting that our ctree be a *decision ctree* with respect to our SBN. The second requirement can be enforced, in addition to using a decision ctree, by restricting the decision ctree to respect a topological ordering. In this case, the child cluster is guaranteed to appear as the right-most leaf in the ctree (which guarantees a conditional ctree). Algorithm 3 provides an algorithm for finding such a ctree, given a cluster DAG and a topological ordering of its clusters. The following proposition summarizes the result.

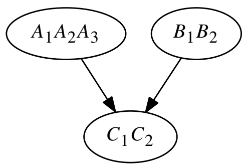
Proposition 3. *The ctree v returned by Algorithm 3 is a decision ctree with respect to the input cluster DAG. Moreover, for each family $\boldsymbol{X} \mid \boldsymbol{P}$ in the cluster DAG, the projection of v onto family $\boldsymbol{X} \mid \boldsymbol{P}$ is a conditional ctree for the family.*

This ctree and its implied conditional ctrees can be used to perform exact inference in an SBN via compilation. However, we must first show how to choose the sub-vtrees over the variables **X** of a cluster, which we do next.

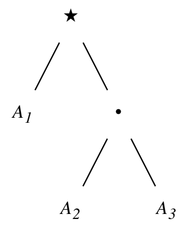
5.5.3 Finding Local Vtrees

Typically, when learning PSDDs from data, one must also learn its vtree [LBB17]. In an SBN, it suffices to learn the conditional PSDDs of its conditional distributions. To provide the best fit, we should learn the corresponding conditional vtrees independently. However, to perform inference, these conditional vtrees must be compatible, as projections of a common global vtree. We show how to achieve this next.

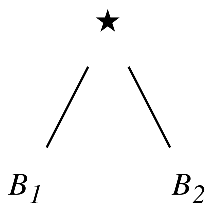
Consider the cluster DAG of Figure 5.4, and its three conditional vtrees. Here, the vtree for cluster **A** is not compatible with the vtree for cluster **C**, as their projections onto variables **A** are different vtrees. More generally, consider a family $\mathbf{X} \mid \mathbf{P}$ and its conditional vtree v. Let u denote the **X**-node of v. Sub-vtree u dictates the probabilistic (PSDD) component of



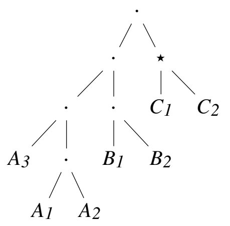
(a) Cluster DAG



(b) Vtree for cluster **A**



(c) Vtree for cluster **B**



(d) Vtree for cluster **C**

Figure 5.4: A cluster DAG and three conditional vtrees for clusters $\mathbf{A} = \{A_1, A_2, A_3\}$, $\mathbf{B} = \{B_1, B_2\}$, and $\mathbf{C} = \{C_1, C_2\}$. $\mathbf{X}$-nodes are labeled with a star.

a conditional PSDD, whereas the sub-vtree outside of u, over $\mathbf{P}$, dictates the logical (SDD) component. That is, the distribution induced by a conditional PSDD depends only on the sub-vtree u over $\mathbf{X}$. The sub-vtree over $\mathbf{P}$ impacts the size of the conditional PSDD but not its distribution. We thus propose to manipulate the conditional vtrees of an SBN so that they become compatible, but for each family $\mathbf{X} \mid \mathbf{P}$ with conditional vtree v and $\mathbf{X}$-node u, we leave the sub-vtree u fixed in v. In this case, the probabilistic (PSDD) components of each conditional PSDD will also be fixed, leaving the conditional distributions invariant.

We propose the following two-step algorithm, that takes as input an SBN whose conditional PSDDs have been learned independently from data, and outputs an SBN with an equivalent joint distribution, that further accommodates exact inference via compilation. Our first step is to obtain a target decision ctree v, where each conditional vtree will be made compatible with v. We first run Algorithm 3 to obtain a decision ctree c for our SBN. For each family $\mathbf{X} \mid \mathbf{P}$, we then replace the leaf cluster $\mathbf{X}$ in ctree c with the $\mathbf{X}$-node of the family's conditional vtree, yielding our target vtree v.

Our second step is to adjust all conditional vtrees so that it is compatible with v. More specifically, for each family $\mathbf{X} \mid \mathbf{P}$, we adjust the logical (SDD) component of its conditional vtree/PSDD, through a process called *re-normalization*.[7] Tree rotation and swap operators

[7]For a family $\mathbf{X}|\mathbf{P}$, let w be its conditional PSDD, and let v' be the projection of decision vtree v onto $\mathbf{X}|\mathbf{P}$. First, we extract the SDD circuit of a conditional PSDD. For both vtrees w and v', we replace the leaf cluster $\mathbf{X}$ with a dummy leaf vtree; we also replace the corresponding PSDD nodes of the conditional PSDD with dummy terminals. We then re-normalize the resulting (algebraic) SDD so that it conforms to

Table 5.2: Compilation versus jointree inference by map size. Size is # of edges. Improvement is jointree over evaluation time.

map size	depth	compilation time (s)			evaluation time (s)			jointree time (s)			improvement
910	6.2	47.422	±	8.708	2.223	±	0.572	7.374	±	1.595	3.317 ×
2,103	7.4	290.919	±	70.580	9.156	±	3.298	27.044	±	2.809	2.953 ×
3,241	8.6	785.877	±	171.873	13.458	±	3.905	55.011	±	2.447	4.087 ×
5,202	9.6	2,003.242	±	424.589	21.492	±	2.783	135.941	±	41.475	6.325 ×
7,621	10.6	3,938.633	±	558.427	26.315	±	5.111	249.658	±	41.422	9.487 ×
9,290	10.8	8,508.526	±	2,778.961	51.645	±	17.204	519.567	±	201.134	10.060 ×
10,500	11.2	12,333.635	±	3,086.726	60.136	±	14.517	607.294	±	173.474	10.098 ×

were previously used to re-normalize an SDD to a new vtree [CD13], where an operation on the vtree implied a corresponding re-factorization of the logical circuit of the SDD.[8] After re-normalization, the resulting conditional PSDDs are now all compatible with each other.

5.6 Experiment

We compare the efficiency of our exact inference algorithm for SBNs, with jointree message-passing using sparse tables [LD03].[9] We evaluate these inference algorithms on an SBN induced from a hmap, as done in Chapter 6. We obtained public map data of San Francisco (SF) from `openstreetmap.org`, and selected 7 increasingly larger regions of SF. We induced a random hmap from each map. For each map size, we generated $5 \times 5 \times 5 = 125$ problem instances: (1) 5 random hmaps, (2) 5 random parameterizations of the SBN, and (3) 5 MPE queries (what is the most likely route between randomly chosen source/destination pairs). In Table 5.2, we report averages and standard deviations for the times to (1) compile the joint

v' instead of w, and replace the dummy terminals with the original PSDD nodes.

[8]To re-normalize an SDD, it also suffices to re-construct the SDD bottom-up for the new vtree, using the SDD's `apply` operator [Dar11], which we did in our experiments.

[9]We first reduce our SBN to a flat factor graph, and induce sparse factors from each conditional PSDD. We pick a jointree structure that reflects the hmap that was used; this allows it to exploit the significant determinism in the CPTs. Otherwise, jointree inference would be intractable for such models.

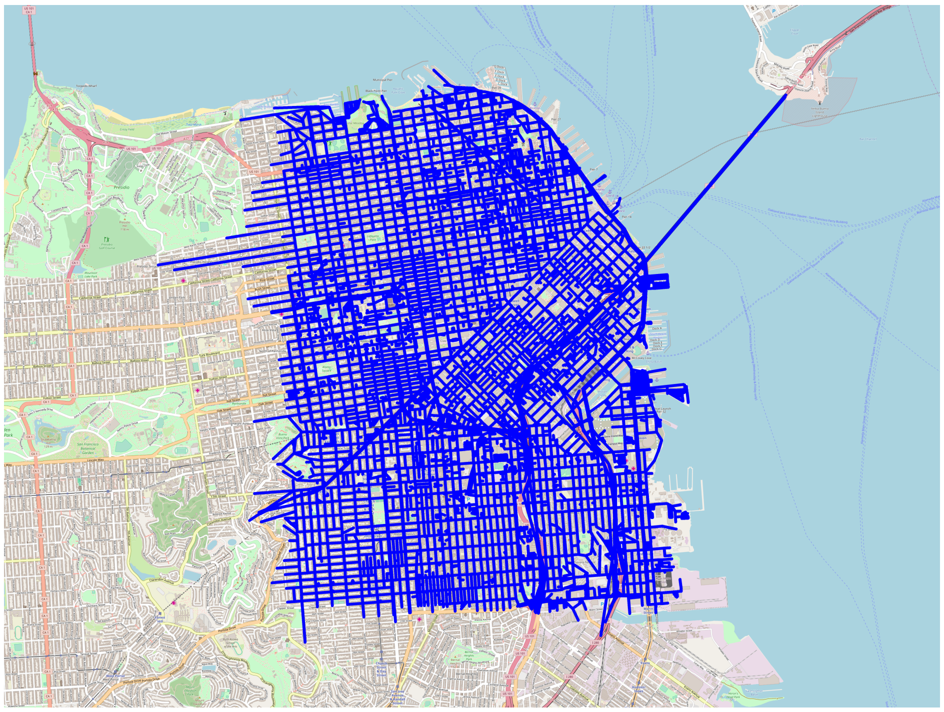


Figure 5.5: The road network of downtown SF.

PSDD (offline step), (2) evaluate the PSDD (online query step), and (3) run the jointree algorithm. For each region, the graph size reported is the number of edges (road segments) in the map, and depth is the average depth of the binary hmap.

The largest map considered had $10,500$ edges; its graph is highlighted in Figure 5.5. On average, the corresponding SBN had 1.7M parameters and the joint PSDD was of size 8.9M (edges), which highlights the scalability of our approach. Next, observe that as we increase the size of the map, evaluation time of the joint PSDD is increasingly more efficient than the jointree algorithm, and over one order-of-magnitude more efficient in the largest graph evaluated. This is in part due to the ability of PSDDs to exploit context-specific independence as well as determinism, whereas sparse tables only take advantage of determinism. While binary hmaps are tractable, we expect the gap between compilation and jointree inference to grow for general SBNs. Finally, we note that compilation time is non-trivial, although this is a one-time cost that is spent offline. When many queries are performed online, the time savings obtained by using joint PSDDs will be a considerable advantage.

5.7 Conclusion

In this chapter, we proposed a polytime multiplication operator for PSDDs. Using this operator, we provided a relatively simple but effective algorithm for compiling probabilistic graphical models into PSDDs. Furthermore, we showed that the same procedure can also be utilized to compile structured Bayesian networks (SBNs) to PSDDs. We highlighted the importance of vtrees for the purposes of inference and separately for learning. Empirically, we showed that inference based on compilation can be an order-of-magnitude more efficient compared to a jointree inference algorithm.

5.A Proofs

Proof of Theorem 4. We first observe that for DNNF circuits f_1 and f_2, checking whether $f_1 \wedge f_2$ is consistent is NP-hard [DM02]. We will now reduce this consistency test to multiplying two decomposable ACs. We first convert each DNNF circuit f_i into an arithmetic circuit $\mathcal{AC}_i$ by replacing or-gates with +-nodes, and-gates with *-nodes, inputs x with λ_x, and true/false with 1/0. The resulting AC is decomposable and satisfies $\mathcal{AC}_i(\mathbf{x}) = 0$ iff f_i outputs 0 on input $\mathbf{x}$ (i.e., $\mathcal{AC}_i(\mathbf{x}) > 0$ iff f_i outputs 1 on input $\mathbf{x}$). Hence, $f_1 \wedge f_2$ outputs 0 on input $\mathbf{x}$ iff $\mathcal{AC}_1(\mathbf{x}) \cdot \mathcal{AC}_2(\mathbf{x}) = 0$. Therefore, $f_1 \wedge f_2$ is consistent iff there exists an input $\mathbf{x}$ with $(\mathcal{AC}_1 \cdot \mathcal{AC}_2)(\mathbf{x}) > 0$, which holds iff the partition function of $\mathcal{AC}_1 \cdot \mathcal{AC}_2$ is greater than 0. The latter condition can be checked in polytime on decomposable ACs. An analogous argument can be used for the multiplication of decomposable and deterministic ACs, using a consistency test on the conjunction of two d-DNNFs [DM02]. □

Proof of Theorem 5. That the AC is decomposable and deterministic follows directly from the definition of a PSDD and its underlying SDD. That the AC represents the distribution induced by the PSDD can be shown using an inductive argument on the structure of the PSDD. For the base case, we have a PSDD literal x and AC indicator λ_x, or a PSDD terminal $\perp$ and AC constant 0, or a PSDD simple or-node and AC $\alpha\lambda_x + (1-\alpha)\lambda_{\neg x}$. The theorem holds for all cases.

For the inductive case, consider an or-node n of the PSDD with elements (p_i, s_i, α_i), and the corresponding $+$-node $\mathcal{AC}_n$ of the AC which has the form $\sum_i \alpha_i \cdot \mathcal{AC}_{p_i} \cdot \mathcal{AC}_{s_i}$. By induction, we assume the PSDD/AC pairs $p_i/\mathcal{AC}_{p_i}$ and $s_i/\mathcal{AC}_{s_i}$ induce the same distribution, i.e., the value of the PSDD is the same as the value of the AC given the same input. Given input $\mathbf{x}$, at most one element (p_i, s_i, α_i) of the PSDD will have its corresponding SDD wire evaluate to 1. The value of the PSDD given input $\mathbf{x}$ is the product of α_i and the values of p_i and s_i. Similarly, the AC has the same non-zero child $\alpha_i \cdot \mathcal{AC}_{p_i} \cdot \mathcal{AC}_{s_i}$ (the others must have value zero by induction). Hence, the distributions of PSDD n and circuit $\mathcal{AC}_n$ are the same. $\square$

Proof of Theorem 6. Let input PSDDs n_1 and n_2 have elements (p_i, s_i, α_i) and (q_j, r_j, β_j), respectively. If vtree v is over variables $\mathbf{X}$, then denote the variables of the left and right children v^l and v^r by $\mathbf{X}^l$ and $\mathbf{X}^r$, respectively. Let Pr_n denote the distribution of a PSDD n. First, for an instantiation $\mathbf{x}^l$ there is a unique p_i and a unique q_j where $\mathbf{x}^l \models p_i$ and $\mathbf{x}^l \models q_j$. Subsequently:

$$\begin{aligned}
&Pr_{n_1}(\mathbf{x}) \cdot Pr_{n_2}(\mathbf{x}) = \Big(Pr_{p_i}(\mathbf{x}^l) \cdot Pr_{s_i}(\mathbf{x}^r) \cdot \alpha_i\Big) \cdot \Big(Pr_{q_j}(\mathbf{x}^l) \cdot Pr_{r_j}(\mathbf{x}^r) \cdot \beta_j\Big) \\
&= (Pr_{p_i}(\mathbf{x}^l) \cdot Pr_{q_j}(\mathbf{x}^l)) \cdot (Pr_{s_i}(\mathbf{x}^r) \cdot Pr_{r_j}(\mathbf{x}^r)) \cdot (\alpha_i \cdot \beta_j) \\
&= \Big(\frac{1}{\kappa_{p_i q_j}} \cdot Pr_{p_i}(\mathbf{x}^l) \cdot Pr_{q_j}(\mathbf{x}^l)\Big) \cdot \Big(\frac{1}{\kappa_{s_i r_j}} \cdot Pr_{s_i}(\mathbf{x}^r) \cdot Pr_{r_j}(\mathbf{x}^r)\Big) \cdot \Big(\kappa_{p_i q_j} \cdot \kappa_{s_i r_j} \cdot \alpha_i \cdot \beta_j\Big)
\end{aligned}$$

where $\kappa_{p_i q_j}$ and $\kappa_{s_i r_j}$ are the normalizing constants of $p_i \cdot q_j$ and $s_i \cdot r_j$, respectively. Let $\kappa = \sum_{ij} \kappa_{p_i q_j} \cdot \kappa_{s_i r_j} \cdot \alpha_i \cdot \beta_j$ denote the normalizing constant of $Pr_{n_1} \cdot Pr_{n_2}$. The above expression corresponds to a PSDD for the desired distribution $\frac{1}{\kappa} \cdot Pr_{n_1} \cdot Pr_{n_2}$, which has elements:

$$\Big(\frac{1}{\kappa_{p_i q_j}} \cdot p_i \cdot q_j, \frac{1}{\kappa_{s_i r_j}} \cdot s_i \cdot r_j, \frac{1}{\kappa} \cdot \kappa_{p_i q_j} \cdot \kappa_{s_i r_j} \cdot \alpha_i \cdot \beta_j\Big)$$

Algorithm 2 recursively constructs this PSDD. The base case used in Line 2.2 is as follows:

$n_1 \backslash n_2$	$\perp$	X	$\neg X$	$X : \beta$
$\perp$	$(\perp, 0)$	$(\perp, 0)$	$(\perp, 0)$	$(\perp, 0)$
X	$(\perp, 0)$	$(X,1)$	$(\perp, 0)$	(X,β)
$\neg X$	$(\perp, 0)$	$(\perp,0)$	$(\neg X,1)$	$(\neg X,1-\beta)$
$X : \alpha$	$(\perp, 0)$	(X,α)	$(\neg X,1-\alpha)$	$(X : \frac{\alpha \cdot \beta}{\kappa},\kappa)$

where $\kappa = \alpha \cdot \beta + (1-\alpha) \cdot (1-\beta)$, and $X : \theta$ represents a simple or-gate over variable X such that the literal X has weight θ and the literal $\neg X$ has weight $1-\theta$.

Let i_v denote a node in input PSDD n_1 normalized for vtree node v and let j_v denote a node in input PSDD n_2 normalized for vtree v. Let $size(i_v)$ be the number of elements for PSDD node i_v (and similarly for j_v). The size of input PSDD n_1 is then $s_1 = \sum_{v,i_v} size(i_v)$ (and similarly for s_2). Due to caching, we invoke the algorithm at most once for each pair of PSDD nodes (i_v, j_v). Moreover, given the Cartesian product on the elements of i_v and j_v, the overall complexity of the algorithm is

$$O(\sum_v \sum_{i_v j_v} size(i_v) \cdot size(j_v)) = O\left(\left[\sum_{v,i_v} size(i_v)\right]\left[\sum_{v,j_v} size(j_v)\right]\right) = O(s_1 s_2).$$

□

Proof of Theorem 7. We first observe that for a d-DNNF circuit f, checking the validity of $\exists X f$ is NP-hard [DM02]. We will now reduce this test to summing out a variable from a deterministic and decomposable AC. We first convert the d-DNNF circuit f into a decomposable and deterministic arithmetic circuit $\mathcal{AC}$ as given in the proof of Theorem 4. Recall that the resulting AC is such that $\mathcal{AC}(\mathbf{x}) = 0$ iff f outputs 0 on input $\mathbf{x}$. Let $\mathbf{Y} = \mathbf{X} \setminus X$. Then $\exists X f$ outputs 0 on input $\mathbf{y}$ iff $(\sum_X \mathcal{AC})(\mathbf{y}) = 0$. Therefore, $\exists X f$ is valid iff $\min_{\mathbf{y}}(\sum_X \mathcal{AC})(\mathbf{y}) > 0$, which can be decided in polytime if $\sum_X \mathcal{AC}$ is decomposable and deterministic. □

Proof of Theorem 8. The proof is constructive. First, we identify a distribution that has no compact PSDD representation for any vtree. We then show that this distribution results from summing out a variable from another distribution that can be represented compactly as a PSDD.

Let PSDDs n_1 and n_2 represent two fully-factorized distributions Pr_1 and Pr_2 over variables $\mathbf{Z}$. Let PSDD a represent the weighted addition $Pr_a = \theta_1 Pr_1 + \theta_2 Pr_2$ where θ_1 and θ_2 are positive weights that sum to one. Let $\mathbf{X}$ and $\mathbf{Y}$ be a partition of variables $\mathbf{Z}$ and consider the conditional distribution $Pr_a(\mathbf{Y} \mid \mathbf{x})$ for some instantiation $\mathbf{x}$. We now have

$$Pr_a(\mathbf{Y}, \mathbf{x}) = \theta_1 Pr_1(\mathbf{Y}, \mathbf{x}) + \theta_2 Pr_2(\mathbf{Y}, \mathbf{x}) = \theta_1 Pr_1(\mathbf{x}) Pr_1(\mathbf{Y}) + \theta_2 Pr_2(\mathbf{x}) Pr_2(\mathbf{Y})$$

$$Pr_a(\mathbf{x}) = \sum_{\mathbf{y}} Pr_a(\mathbf{xy}) = \sum_{\mathbf{y}} \big[\theta_1 Pr_1(\mathbf{xy}) + \theta_2 Pr_2(\mathbf{xy})\big] = \theta_1 Pr(\mathbf{x}) + \theta_2 Pr(\mathbf{x}).$$

Hence,

$$\begin{aligned} Pr_a(\mathbf{Y} \mid \mathbf{x}) &= \frac{Pr_a(\mathbf{Y}, \mathbf{x})}{Pr_a(\mathbf{x})} = \frac{\theta_1 Pr_1(\mathbf{x}) Pr_1(\mathbf{Y}) + \theta_2 Pr_2(\mathbf{x}) Pr_2(\mathbf{Y})}{\theta_1 Pr_1(\mathbf{x}) + \theta_2 Pr_2(\mathbf{x})} \\ &= \tau_{\mathbf{x}} Pr_1(\mathbf{Y}) + (1 - \tau_{\mathbf{x}}) Pr_2(\mathbf{Y}) \end{aligned}$$

where

$$\tau_{\mathbf{x}} = \frac{\theta_1 Pr_1(\mathbf{x})}{\theta_1 Pr_1(\mathbf{x}) + \theta_2 Pr_2(\mathbf{x})} = \frac{1}{1 + \frac{\theta_2 Pr_2(\mathbf{x})}{\theta_1 Pr_1(\mathbf{x})}}.$$

The conditional distribution $Pr_a(\mathbf{Y} \mid \mathbf{x})$ is then a weighted sum of the fully factorized distributions $Pr_1(\mathbf{Y})$ and $Pr_2(\mathbf{Y})$, where the weight $\tau_{\mathbf{x}}$ is a function of the instantiation $\mathbf{x}$. Assume that $Pr_1(\mathbf{Y})$ and $Pr_2(\mathbf{Y})$ are distinct. First, note that any distinct weight $\tau_{\mathbf{x}}$ yields a distinct conditional distribution $Pr_a(\mathbf{Y} \mid \mathbf{x})$. Second, with the appropriate parameterization of Pr_1, Pr_2, we can guarantee that the weight $\tau_{\mathbf{x}}$ is distinct for all distinct instantiations $\mathbf{x}$.[10] Since we have $2^{|\mathbf{X}|}$ distinct instantiations $\mathbf{x}$, we have $2^{|\mathbf{X}|}$ distinct conditional distributions $Pr_a(\mathbf{Y} \mid \mathbf{x})$.

A vtree node v_i on the right most path will partition variables $\mathbf{Z}$ into $\mathbf{X}$ and $\mathbf{Y}$, where $\mathbf{Y}$ are the variables inside vtree v_i. Any distinct conditional distribution $Pr_a(\mathbf{Y} \mid \mathbf{x})$ must have a distinct PSDD node normalized for vtree v_i, leading to $2^{|\mathbf{X}|}$ such nodes in the above construction. Hence, the PSDD for Pr_a is exponentially large. This is analogous to the [SW93] construction and bound for OBDDs.

Consider now the distribution $Pr_c(U, \mathbf{X}, \mathbf{Y})$ such that $Pr_c(u, \mathbf{x}, \mathbf{y}) = \theta_1 Pr_1(\mathbf{X}, \mathbf{Y})$ and $Pr_c(\overline{u}, \mathbf{x}, \mathbf{y}) = \theta_2 Pr_2(\mathbf{X}, \mathbf{Y})$, where Pr_1, Pr_2 and θ_1, θ_2 are as given above. Distribution Pr_c

[10]Note $\frac{Pr_2(\mathbf{x})}{Pr_1(\mathbf{x})} = \prod_{i \in I_{\mathbf{x}}} \frac{q_i}{p_i} \prod_{j \notin I_{\mathbf{x}}} \frac{1-q_j}{1-p_j}$ where $I_{\mathbf{x}}$ is the set of indices i where X_i is set to true by $\mathbf{x}$. Each $I_{\mathbf{x}}$ is unique for each distinct $\mathbf{x}$, so each $\tau_{\mathbf{x}}$ is unique if we set $\frac{1-p_i}{1-q_i} = \frac{1}{2}$ and $\frac{p_i}{q_i}$ to a unique prime for all i.

can be represented as a PSDD whose size is linear in $n = |\mathbf{Z}|$. Summing-out variable U from Pr_c results in the distribution $Pr_a(\mathbf{X}, \mathbf{Y})$, which has an exponentially large PSDD for any vtree (as shown above). $\square$

Proof of Theorem 9. The proof (sketch) is constructive. First, assume we have a jointree for the factors with width w (such a jointree must exist by the definitions of jointree and treewidth). We will now construct a vtree recursively from the given jointree, assuming that we have selected some arbitrary cluster $\mathbf{C}$ as the jointree root. The base case is for a jointree with only one cluster $\mathbf{C}$, leading to a right-linear vtree over the variables of $\mathbf{C}$ (that is, a vtree in which the left child of each internal node is a leaf). For the inductive case, remove root $\mathbf{C}$ from the jointree, leading to a number of disconnected trees t_i and select the neighbor of $\mathbf{C}$ in each t_i as the root for t_i. Construct a vtree v_i recursively from each t_i and its root. Connect these vtrees arbitrarily into a vtree $v_{\mathbf{C}}$. The final vtree will be constructed in a right-linear fashion, first with each of the variables $\mathbf{C}$ and then with the node $v_{\mathbf{C}}$ last. This construction leads to a decision vtree as defined in [OD14], where factors play the role of CNF clauses.

Suppose now that we construct a PSDD for the given factors using the above vtree. One can show that if vtree node v was added when processing cluster $\mathbf{C}$, then the PSDD will have at most $2^{|\mathbf{C}|}$ nodes normalized for v. Moreover, one can show that each PSDD node will have two elements. Hence, the size of resulting PSDD will be $O(m \cdot \exp(w))$. $\square$

CHAPTER 6

Case Study: Modeling Routes using SBN

In this chapter, we illustrate the promise of SBNs by providing a case study in modeling a distribution over routes on a map. The results discussed in this chapter appeared in [CSD17], [SCD18] and [SGD19].

6.1 Introduction

Route distributions are of great practical importance as they can be used to estimate traffic jams, predict routes, and even predict the impact of interventions, such as closing certain streets on a road network. In this chapter, we consider a probabilistic route model that is based on a hierarchical decomposition of the map. Given a hierarchical map, we can in turn induce a structured Bayesian network (SBN), a recently proposed class of probabilistic graphical models. By enforcing a special partitioning scheme, the SBN, that represents the route model, is shown to be tractable, and route queries can be exactly computed using the inference method that is described in Chapter 5..

This chapter is organized as follows. In Section 6.2, we review the problem of learning distributions over routes, and review representing routes using propositional variables in Section 6.3. We first introduce a hierarchical partition of a map in Section 6.4. From this hierarchical partition, we present a probabilistic model of routes in Section 6.5 using the SBN framework. Further, we exploit the tractable subclass of the model by using a particular map partition scheme that is described in Section 6.6. It follows in Section 6.7 that this special partition scheme can be learned from data. In Section 6.8, we empirically study our route model using the learning algorithm and demonstrates different interesting queries on

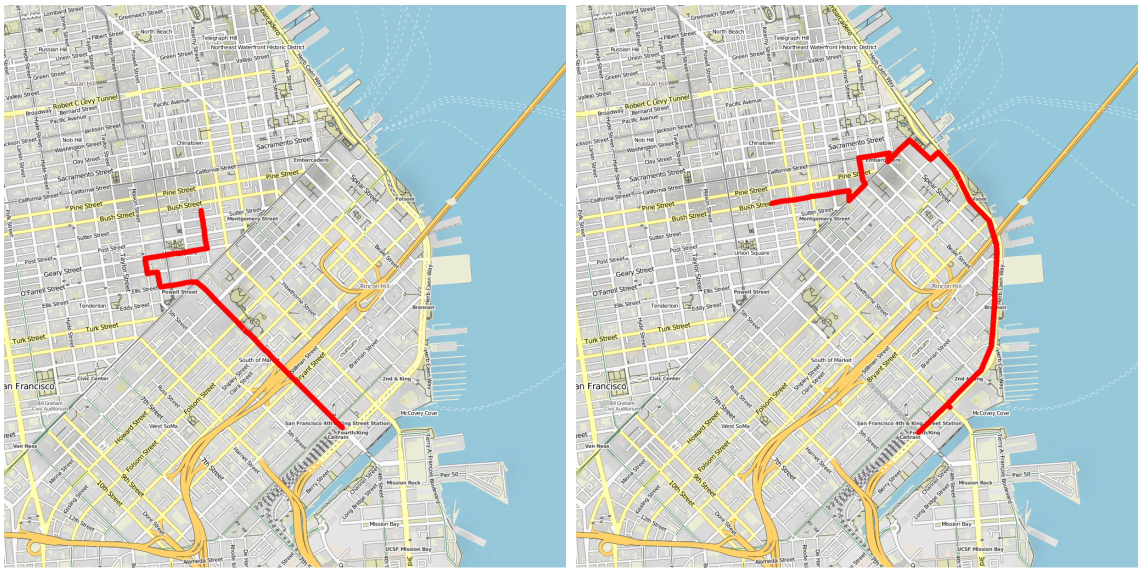


Figure 6.1: Routes between the San Francisco Caltrain station at 4th & King Street and the entrance to Chinatown.

the route distribution. We conclude in Section 6.9.

6.2 Distributions over Routes: A Primer

Given the ubiquity of GPS navigators, there is a tremendous amount of data available about drivers and the routes that they take. In this chapter, we seek to learn distributions over these routes, which can be used to estimate traffic jams, predict routes, and even predict the impact of interventions (e.g., a city planner may want to know the impact of closing certain streets for a parade).

Take for example [PSG09], which considers a dataset consisting of more than 380,000 taxi trips in San Francisco. Each route consists of a sequence of latitude/longitude pairs, along with a timestamp. For example, the trace depicted in Table 6.1 corresponds to a single taxi trip. The time stamp is given in UNIX time. The above trip lasted at least 770 seconds. Consider Figure 6.1, which depicts two different trips in San Francisco, between the San Francisco (Caltrain) station and the entrance to Chinatown (Dragon's gate). One route is relatively direct, while the other is more "scenic.". Our goal again is to, given such

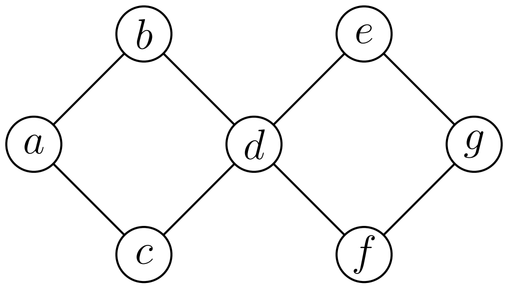
Table 6.1: Trace of a taxi trip

`latitude`	`longitude`	`time_stamp`
37.79326	-122.40061	1213071358
37.79274	-122.40270	1213071426
37.79219	-122.40901	1213071478
37.79641	-122.41009	1213071539
37.79894	-122.41064	1213071598
37.80082	-122.41073	1213071664
37.80274	-122.41130	1213071692
37.80487	-122.41324	1213071752
37.80526	-122.41667	1213071823
37.80364	-122.42846	1213071933
37.80111	-122.42785	1213071971
37.80107	-122.42795	1213072031
37.79975	-122.43016	1213072092
37.79988	-122.43109	1213072128

a dataset, learn a corresponding distribution over routes.

One approach to learning a distribution over routes is to treat a given map as a graph: each street on a map corresponds to an edge on the graph, with each intersection corresponding to a node. A distribution over routes on a map then corresponds to a distribution over paths on a graph. In this chapter, we focus on *simple* paths on a graph, i.e., paths that do not visit the same node twice.

One naive way to learn a distribution over routes is to enumerate all routes σ and then estimate a probability $Pr(\sigma)$ for each, by counting how many times each route σ appears in a dataset, and normalizing the counts so that $\sum_\sigma Pr(\sigma) = 1$. Consider the following graph (i.e., map):



There are four routes connecting nodes a and g. The following table depicts a distribution over these routes:

route σ	$Pr(\sigma)$
(a, b, d, e, g)	0.1
(a, b, d, f, g)	0.2
(a, c, d, e, g)	0.3
(a, c, d, f, g)	0.4

Here, it is simple to perform a query; e.g., the most likely route from a to g that first visits b is the route (a, b, d, f, g) with probability 0.2. However, such a tabular representation quickly becomes infeasible as the graph becomes larger. Consider $n \times n$ grids (with n^2 nodes). The following table counts all paths from the upper-left to the bottom-right corner (which is just a subset of all possible routes).

size of grid	# of routes
2×2	2
3×3	12
4×4	184
5×5	8,512
6×6	1,262,816
7×7	575,780,564
8×8	789,360,053,252
9×9	3,266,598,486,981,642
10×10	41,044,208,702,632,496,804

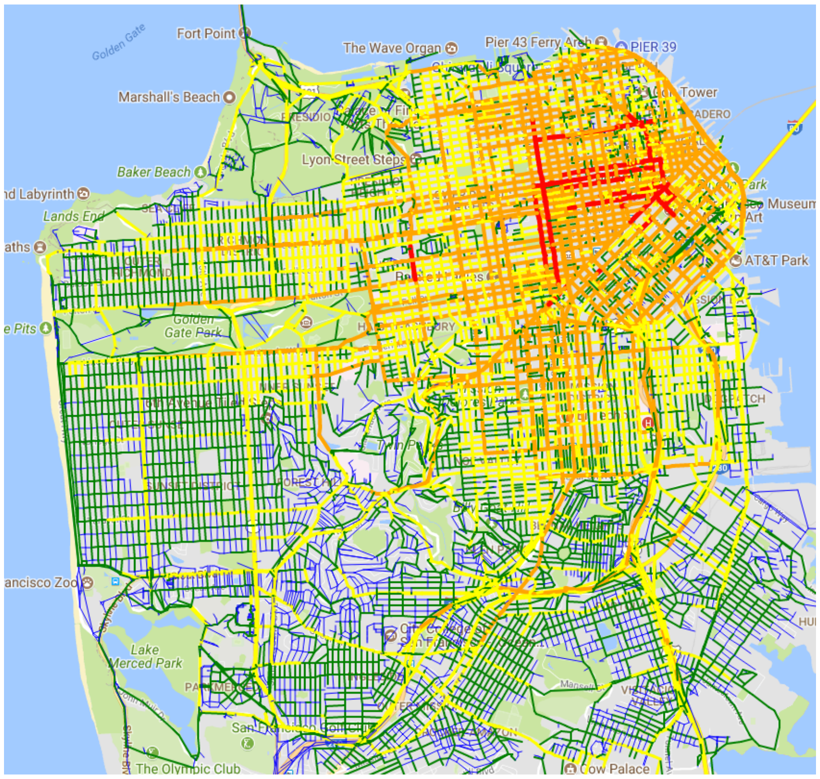


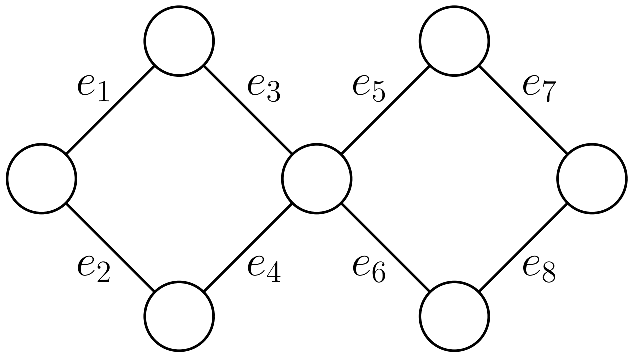
Figure 6.2: Estimating the probability that a street is used in a route, based on on taxi cab traces in San Francisco. Color indicates the popularity of a street: blue ($<$ 10 routes), green (10's of routes), yellow (100's of routes), orange (1,000's of routes), and red (10,000's of routes).

As is evident in the above table, we would quickly run out of memory to store such a tabular representation. Moreover, we would need a correspondingly infeasible amount of data to learn such a distribution.

Figure 6.2 depicts another (naive) alternative. Here, we have counted how many times each street was used by a route in the dataset. This corresponds to a fully-factorized probabilistic model, where each random variable represents an edge of the graph, which is true if the edge was used in a route, and false otherwise. Such a model provides some useful information (such as the popularity of a street), but is useless for any query that involves more than one street (e.g., predicting a route). Among probabilistic models, (hidden) Markov models are popular for modeling routes; see, e.g. [SBZ06, Kru08].

6.3 Representing the Space of Routes

Consider the following graph G.

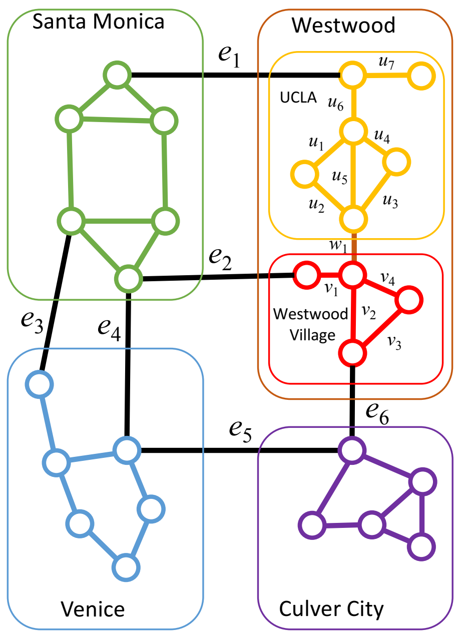


Here, we have labeled each edge of G by a label e_i. For each edge e_i, we assume a Boolean variable X_i that represents whether edge e_i is used on a path or not (i.e., it is true or false). Let $\mathbf{X} = \{X_1, \ldots, X_8\}$ denote the set of edge variables, and let $\mathbf{x}$ denote a corresponding instantiation. An instantiation $\mathbf{x}$ then corresponds to a selection of edges e_i when the corresponding variables X_i have been set to true. Some instantiations correspond to a route and other do not. For example, an instantiation $\mathbf{x} = (\mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{false})$ corresponds to the route (e_1, e_3, e_5, e_7), whereas instantiation $\mathbf{x} = (\mathsf{true}, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{true})$ selects the edges e_1, e_2, e_7, e_8 which is not a route. A probability distribution $Pr(\mathbf{X})$ is called a *route distribution* iff it assigns a zero probability to every instantiation $\mathbf{x}$ that does not correspond to a route.

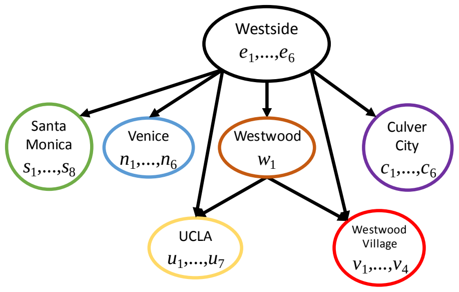
6.4 Hierarchical Routes

Consider Figure 6.3a which depicts a simplified graph representing the Los Angeles (LA) Westside. Here, the nodes (intersections) of the LA Westside have been partitioned into regions. The LA Westside is first partitioned into four sub-regions: Santa Monica, Westwood, Venice and Culver City. Westwood is further partitioned into two smaller sub-regions: UCLA and Westwood Village. This partitioning is an example of a *hierarchical map.*

A hierarchical map allows us to abstract the notion of a route. That is, we can think of an abstract route as a route between regions (those of the partition). We then refine this



(a) Map



(b) cluster DAG

Figure 6.3: A hierarchical map of neighborhoods in the Los Angeles Westside, and the corresponding cluster DAG. Each neighborhood and the roads within are depicted using colors, while roads connecting regions are depicted in black. The roads of Santa Monica, Venice and Culver City are unlabeled in the map, for clarity.

route by recursively planning the routes in each region, under the constraint that they are consistent with the abstract route we found between regions. For example, in Figure 6.3a, if we want to go from Venice to Westwood, we may first decide to use edge e_4 to go from Venice to Santa Monica, and then use edge e_1 to go from Santa Monica to Westwood. Next, we find a route in Venice to edge e_4, then a route through Santa Monica from edge e_4 to e_1, and finally a route in Westwood from e_1 to the destination (we can then recursively find routes between UCLA and Westwood Village).

We assume simple routes (no cycles) at each level of the hierarchy by excluding routes that enter or leave the same region more than once. Route that satisfies this property is called hierarchical simple route. A simple route on the original map may not be hierarchical simple. For example, although a route with edges $e_1, u_6, u_5, w_1, v_1, e_2$ corresponds to a simple path on the map, it is not hierarchical simple route, because it re-enters the region Santa Monica.

The route enters the region using edge e_6 after it has left earlier using edge e_1. Whether requiring a route to be hierarchical simple constitutes a good approximation depends on the hierarchical decomposition used and corresponding queries. We will elaborate on learning a good hierarchical decomposition in Section 6.7.

6.5 Modeling Hierarchical Routes with SBNs

A hierarchical decomposition of a map can be modeled using a cluster DAG as follows; see Figure 6.3. Each internal cluster represents a region, with its variables being the set of edges that cross between its sub-regions. In Figure 6.3b, the root cluster represents the Los Angeles Westside region and its variables represent edges $e_1, \ldots, e_6$ that are used to cross between its four sub-regions. Each leaf cluster represents the set of edges that are strictly contained in that sub-region. The parent-child relationship in the cluster DAG is based on whether an edge in the parent region can be used to enter the sub-region represented by the child cluster. For example, the UCLA region has Westside as a parent since edge e_1 can be used to enter the UCLA region directly from Santa Monica. The main point here is that a cluster DAG can be automatically generated from a hierarchical map.

The use of hierarchical maps implies key independence assumptions that are made explicit by the corresponding cluster DAG. In particular, a hierarchical map implies the following. Once we know how we are entering a sub-region R, the route we take inside that sub-region is independent of the route we may take in any other sub-region that is not a descendant of R. For example, the route we take inside the UCLA region (edges $u_1, \ldots, u_7$) is independent of the route used in any other region, once we know how we plan to enter or exit the UCLA region (edges w_1 and $e_1, \ldots, e_6$).[1] Such independencies can be easily read off the cluster DAG using the Markovian assumption of Bayesian networks.

We now get to the final part of using structured Bayesian networks for representing and

[1]In this particular case, only w_1 and e_1 are relevant to how we enter or exit the UCLA region. However, this independence is only visible in the conditional PSDD for the UCLA region as it is implied by the conditional constraints not the cluster DAG.

learning route distributions. To induce a distribution over routes in a hierarchical map, one needs to quantify the corresponding cluster DAG using conditional PSDDs. That is, for each cluster with variables $\mathbf{X}$, and whose parent clusters have variables $\mathbf{P}$, one needs to provide a conditional PSDD for $\mathbf{X} \mid \mathbf{P}$. As discussed earlier, this conditional PSDD will be based on conditional constraints for $\mathbf{X} \mid \mathbf{P}$ which define the underlying conditional SDD. The conditional PSDD is then obtained from this SDD by learning parameters from data.

The conditional constraints of a cluster can also be generated automatically from a hierarchical map and are of two types. The first type of conditional constraints rules out disconnected routes in cluster $\mathbf{X}$ (does not depend on the state of parent cluster). The second type rules out routes in region $\mathbf{X}$ which are no longer possible given how we enter or exit that region (depends on the state of parent cluster). Clearly, the root cluster only has constraints of the first type.

6.6 Binary Hierarchical Map

We identify a tractable sub-class of SBNs, which can be compiled into joint PSDDs with only polynomial size—namely those that correspond to *binary hierarchical maps.* This class of SBNs are of practical interest, as they are inspired from an application of SBNs for modeling distributions over routes on a map, or equivalently, simple paths on a graph [Zhe15].

In a *binary hmap*, regions are recursively split into two sub-regions. Such maps have three key properties, that lead to its tractability: (1) the simple-path constraints for interior nodes are trivial to compile,[2] (2) the number of parent instantiations that we need to consider in a conditional PSDD is quadratic in the number of edges crossing into the region,[3] and (3) if the simple-path constraints of a leaf region are too hard to compile, we can make the map deeper until they are compilable. With a binary hmap, we obtain the following *polynomial*

[2]A path consists of crossing from one region to another using a single edge, or not at all, because of the simple path assumption (in a simple path, we cannot visit the same region twice).

[3]Due to the simple-path constraint, a path cannot re-enter a region once it has exited it. Hence there are only a quadratic number of way to enter and exit a region to consider (at most two incident edges can be used).

bound on the size of its joint PSDD.

Theorem 10. *Consider a binary hmap with t nodes, where k is the maximum number of edges assigned to a cluster and n is the maximum number of edges that cross into a region. Let m denote the size of the largest PSDD of any leaf region. The size of the joint PSDD is $O(t \cdot n^2 \cdot (m + k))$.*

6.7 Learning Binary Hierarchical Maps

In this section, we consider how to learn a binary hmap, and hence, the structure of an SBN.

Random Binary Hmaps. Consider the following simple algorithm for inducing a random binary hmap, based on recursively decomposing a map into two regions. First, pick two seed nodes a and b of a map, where a will belong to one region and b will belong to the other. Each region alternates between absorbing a neighboring node into their region (if possible), until all nodes are absorbed. A deeper hmap can then be produced by recursing on the sub-regions. We typically recurse until each leaf region is small enough to be compilable to an SDD.

Learning Hmaps from Data. We next propose a heuristic for learning a binary hierarchical map from a dataset consisting of routes. Consider a popular road on a map which is used by many routes in the data. A random partitioning of the map may put one intersection (node) of the road in one region, the next intersection in another region, and the third intersection in the same region as the first. Hence, a route on this road would exit the first region, enter the second, and then re-enter the first. Such a route is not simple relative to an hmap as it visits the same region twice. This assumption was introduced by , and is important for guaranteeing tractability as we discussed in the previous section.

Hence, we propose a heuristic that tries to avoid situations like the above, for learning a binary hmap from data. Our approach is bottom-up.[4] Intuitively, we want to cluster nodes together if they are commonly used by the same route. First, we assign each node to its

[4]Essentially, learning a binary hmap is a type of hierarchical clustering. See, e.g., [Mur12] which discusses both bottom-up (agglomerative) and top-down (divisive) clustering.

own region. Next, we have an edge between regions if there is a street connecting them, and we give that edge a weight based on the number of routes in the data that crosses from one region to the other. We then find a maximum weight matching[5] and merge each of the paired regions. We update the scores between regions and repeat, until we obtain a single cluster. Next, from the clustering, we want to extract an hmap whose leaf regions are as large as possible, given an upper limit. Hence, to obtain our final hmap, we navigate the clustering in depth-first fashion until we find the first node under our limit which we take as an hmap leaf. We then backtrack and continue until we have picked all of our leaves.

6.8 Experiments

Learning Hierarchical Maps. We now evaluate the algorithm proposed in Section 6.7 for learning binary hmaps, using a route prediction task that we describe next. We first took the region of SF covering 910 edges from Table 5.2. We took the `cabspotting` dataset of GPS traces collected from taxicab routes in SF [PSG09]. Using the map-matching API of the `graphhopper` package, we projected GPS traces onto the map. We used 8,196 routes from this dataset to learn the structure and parameters of our SBNs (using Laplace smoothing).[6] We learned a binary hmap for an SBN using our proposed heuristic and also using a random binary hmap, both described in Section 6.7. We used another 128 routes as test routes to perform route prediction: given a source, destination and a partial trip so far (half the trip), what is the most likely completion? This is an MPE query on a PSDD, which we computed using the inference algorithm based on the PSDD multiply operator, proposed in Chapter 5.

For each route in the test set, we measure its similarity with the route predicted from the PSDD, using three metrics: (1) dissimilarity in segment number (DSN), which counts the proportion of non-common road segments between the true and predicted route, (2)

[5]We used the `networkx` python module.

[6]A route dataset may have paths that are not simple, or paths that do not respect the binary hmap assumption (i.e., visits the same region twice). We can still utilize such routes for training, which helped in our experiments. First, we project each route in the training set onto each family of the SBN—multiple paths through the same region becomes a set of independent paths. A projected route may still not be simple; in this case, we segment it further into sub-paths that are simple.

Hausdorff distance [GNG14], which matches all points in one path with the closest point in the other path, and reports the largest such match (normalized by the true trip length), and (3) the difference between trip lengths (normalized by the true trip length). Note that each metric has its own advantages and disadvantages. Further, we expect better hmaps to provide more accurate route predictions.[7]

We evaluate the quality of routes predicted by each of the two SBNs in the following table:

hmap	DSN	Haus.	trip length	# bad routes
random	0.300	0.120	0.147	3,833 / 59
heuristic	**0.250**	**0.089**	**0.076**	**2,791 / 41**

Each entry is an average over 10 runs with randomly sampled training and testing sets (of size 8,196 and 128) from the 31,175 cabspotting routes inside the region. We see that for all three metrics, our heuristic learns a binary hmap with much higher predictive accuracy. For example, the predictions from the heuristic hmap had half the error compared to a random hmap, in terms of trip length. In the last column, we consider how many simple routes become invalid in the binary hmap, as discussed in Section 6.7. Invalid routes visit the same region twice in the hmap—such routes have probability zero in the SBN. We separately report the number of invalid routes in the training and testing sets. The random binary hmap has 137% more invalid routes, indicating that our heuristic is effective at lowering the number of invalid routes that result in a hierarchical decomposition.

Route Classification. We report results on route classification in Figure 6.4. We consider two classes of routes from two datasets: (1) the `cabspotting` dataset (Taxi), and (2) a simulated dataset collected by querying Google Maps Directions API with source/destination pairs (Google).[8] We took $2^{15} = 32,768$ and $2^{12} = 4,096$ routes from each dataset for train-

[7]Note that standard metrics based on test-set likelihood are difficult to apply. Our model assumes paths are simple across regions, so routes violating this assumption have zero probability. Prior empirical comparisons with SBNs considered, instead of likelihood, domain-specific tasks such as next-turn prediction [Kru08].

[8]In particular, we initially took the map of size 5,202 from Table 5.2, and from the `cabspotting` dataset, we took all 172,265 routes strictly contained in the map. For each route, we took the source, destination, day-of-week and time-of-day, which we used to request a corresponding route from Google Maps.

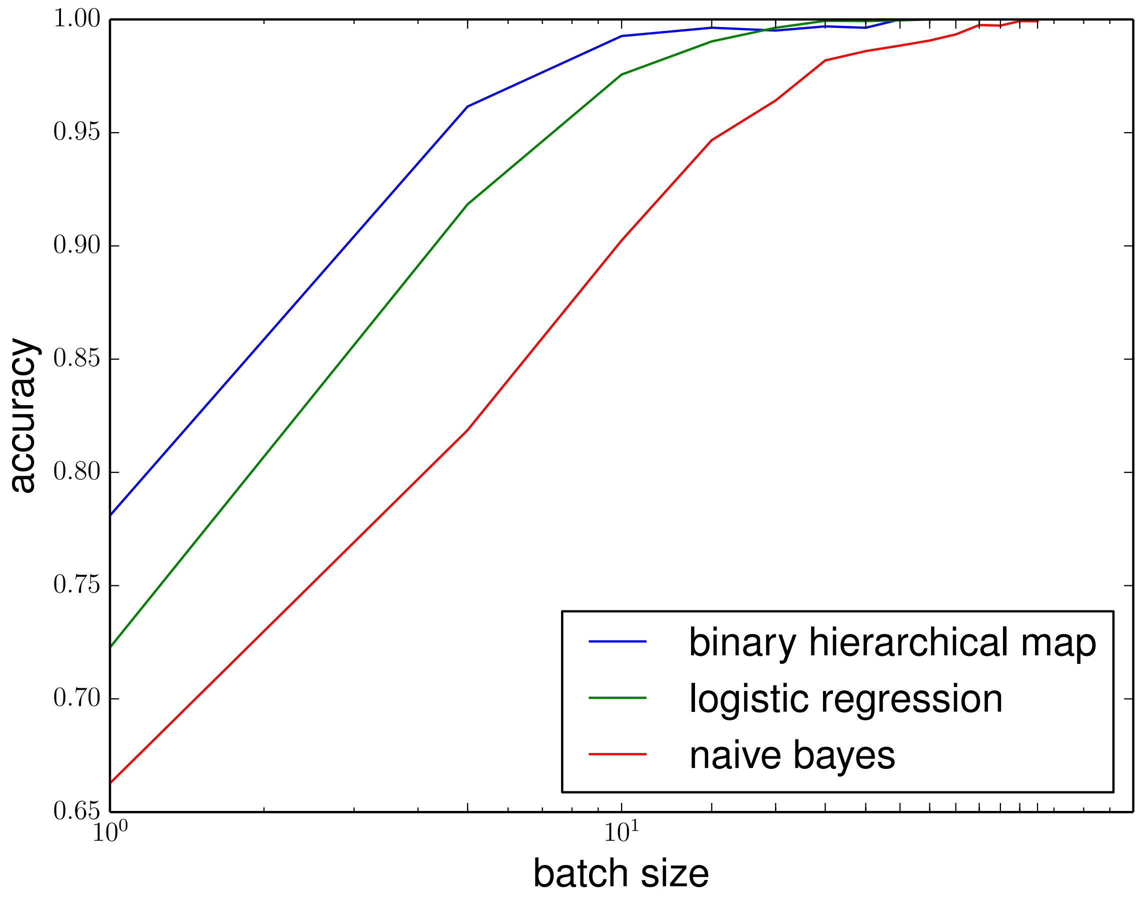


Figure 6.4: Route classification.

ing/testing. We took a map of size 5,374 edges and learned a binary hmap, as in Section 6.7; the SBN had $737,928$ parameters. We trained two sets of SBN parameters (using Laplace smoothing), one for each dataset, yielding a (structured) naive Bayes classifier [CTD16].[9] The Taxi dataset was collected in 2008, which predates the proliferation of GPS navigators. Hence, we view our Google vs. Taxi classifier as discriminating between drivers with/without GPS navigation, or alternatively, Uber drivers (with) vs Taxi drivers (without). Figure 6.4 summarizes our results, where we compare against (1) an (unstructured) naive Bayes classifier (2) and a logistic regression classifier. Both uses each edge as a binary feature (each edge is either present or absent). On the x-axis, we provide each classifier with a batch of routes of increasing size, from 1 to 60. Each batch of size x is a set of x routes of the same

[9]If a train/test route is invalid (visits the same region twice), then it has probability zero in the SBN. In this case, we segment the route into multiple valid routes, as in Footnote 6.8, for training. For testing, we observe each route segment as a feature in the structured naive Bayes classifier.

type—the idea is that the one can better distinguish a driver as an Uber or a taxi driver, as more routes from the same driver are provided. As we give each classifier more routes from the same type of driver, we achieve higher accuracy (as expected), converging to 100% accuracy. Clearly, our SBN (using a binary hmap) is superior to logistic regression, which in turn is superior to naive Bayes.

6.9 Conclusion

Given a map, we showed how to construct a hierarchical decomposition, which can be used to learn a structured Bayesian network (SBN) from GPS data. We studied binary hierarchical maps, which we used to identify a tractable sub-class of SBNs, where exact inference can be performed efficiently. We also studied the problem of learning the structure of the tractable sub-class. Empirically, our approach scales to a large map, and can provide accurate predictions on different tasks, including the route completion and route classification.

6.A Proofs

To prove Theorem 10, we provide a construction of the joint PSDD in this section.

For a given region, we refer to its *external* edges as those edges that have one endpoint inside the region and the other endpoint outside the region. External edges are the edges that are used to enter and exit a region. Further, a path is *simple* if it does not visit the same node twice. We say that a path is *hierarchically simple* if it does not visit the same region twice, in the hmap. In an SBN of a binary hmap, all paths must be hierarchically simple; otherwise, they have probability zero.

Consider a leaf node c in a binary hmap. We want the PSDDs representing routes inside this region. Under the hierarchical simple-path assumption, at most two external edges will be used—we cannot visit the same region twice. At the most, we can enter and exit a region.

When exactly two external edges e_1 and e_2 are used, we want a PSDD over all simple paths that connect to both e_1 and e_2 in the region c. We refer to this PSDD as $\textsf{non-terminal}_c(e_1, e_2)$, because all paths must pass through the region c. When exactly one external edge e is used, we want a PSDD over all simple paths that start at edge e and then end inside region c. We refer to this PSDD as $\mathsf{terminal}_c(e)$, because all paths terminate in region c. Finally, if no external edge is used, we want a PSDD over all simple paths strictly contained inside the region. We refer to this PSDD as $\mathsf{internal}_c$. In PSDDs $\mathsf{terminal}_c(e)$ and PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$ that connect to the same endpoint inside the region, we allow for an empty path inside the region. Further, we assume that all PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$, $\mathsf{terminal}_c(e)$, and $\mathsf{internal}_c$ have a bounded size. This can be ensured by using a binary hmap that is deep enough to have small enough regions. In our experiments, we used the GRAPHILLION package[10] to compile regions into ZDDs, which are then converted to SDDs.

Consider an internal node c in a binary hmap, which has a left child region l and a right child region r. Each internal node c is itself a binary hmap, rooted at c. It represents a map consisting of all nodes and edges inside the corresponding binary hmap. As we did previously, we will construct PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$, $\mathsf{terminal}_c(e)$ and $\mathsf{internal}_c$ over the different types of routes implied by the selected external edges. These PSDDs can be specified using the PSDDs of its left and right child sub-regions.

Suppose we have external edges $e_1, \ldots, e_n$, and the edges $m_1, \ldots, m_k$ that cross between the left and right sub-regions. Let empty_c denote the PSDD for c containing no routes (all edges must be set to false). Consider the PSDD $\mathsf{internal}_c$. An internal route is either (1) strictly contained in the left region, (2) strictly contained in the right region, or (3) it crosses the two regions using exactly one edge m_i. Thus, the corresponding SDD has the elements:

$$\begin{array}{l}
\mathsf{internal}_l, \mathsf{empty}_r \\
\mathsf{empty}_l, \mathsf{internal}_r \\
\mathsf{terminal}_l(m_1), \mathsf{terminal}_r(m_1) \\
\vdots \\
\mathsf{terminal}_l(m_k), \mathsf{terminal}_r(m_k)
\end{array}$$

[10]`https://github.com/takemaru/graphillion`

By the hierarchical simple-path assumption, the path between the left and right regions must consist of a single edge.

Consider the PSDDs $\textsf{terminal}_c(e)$. Suppose that e connects to a node in the left child region (the case for the right child is symmetric). There are two cases: (1) the path stays in the left, or (2) the path crosses into the right using one edge m_i. This PSDD has an SDD with the elements:

$$\begin{aligned}
&\textsf{terminal}_l(e), \textsf{empty}_r \\
&\textsf{non-terminal}_l(e, m_1), \textsf{terminal}_r(m_1) \\
&\vdots \\
&\textsf{non-terminal}_l(e, m_k), \textsf{terminal}_r(m_k)
\end{aligned}$$

Consider the PSDDs $\textsf{non-terminal}_c(e_1, e_2)$. Suppose that e_1 connects to the left child region and that e_2 connects to the right child region (the reverse case is symmetric). Here, the path must cross from the left to the right using one edge m_i. This PSDD has an SDD with the elements:

$$\begin{aligned}
&\textsf{non-terminal}_l(e_1, m_1), \textsf{non-terminal}_r(m_1, e_2) \\
&\vdots \\
&\textsf{non-terminal}_l(e_1, m_k), \textsf{non-terminal}_r(m_k, e_2)
\end{aligned}$$

If e_1 and e_2 both connect to the same region, say the left one, then the path must stay inside the left region. We have an SDD with a single element:

$$\textsf{non-terminal}_l(e_1, e_2), \textsf{empty}_r.$$

To count the total size of the joint PSDD, we count the number of PSDD nodes that we constructed, and also count the size of each node (number of elements). Moreover, we do not count elements with a false sub in our PSDD, which are not needed to represent the distribution. First, for a leaf node in the binary hmap, if n is the number of its external edges, then we have $O(n^2)$ distinct PSDDs, each having a bounded size m. If there are t nodes in the binary hmap, then there are $O(t)$ leaf nodes. Hence, the total size of the leaf

PSDDs is $O(t \cdot n^2 \cdot m)$. Second, for an internal node in the binary hmap, if n is the number of its external edges, then we have $O(n^2)$ PSDD nodes. If k is the number of edges that cross between the left and right sub-regions, then the PSDD node has $O(k)$ elements. Thus, the number of PSDD nodes for internal nodes in the binary hmap is $O(t \cdot n^2)$ and their aggregate size is $O(t \cdot n^2 \cdot k)$. Thus the total size of the joint PSDD is $O(t \cdot n^2 \cdot m + t \cdot n^2 \cdot k)$.

CHAPTER 7

Learning Local Structure from Data

Local structure such as context-specific independence (CSI) has received much attention in the probabilistic graphical model (PGM) literature, as it facilitates the modeling of large complex systems, as well as for reasoning with them. Previously, we have shown that conditional PSDDs can extract CSI from the logical constraints. In this chapter, we provide a new perspective on how to learn CSIs solely from data. The results in this chapter appeared in [SCD20].

7.1 Introduction

Context-specific independence (CSI) is a type of local structure that facilitates the modeling of large and complex systems, by allowing one to represent in a succinct way conditional distributions that would otherwise be infeasible to represent [BFG96]. Further, local structure such as context-specific independence can be exploited by modern classes of inference algorithms to perform reasoning in Bayesian networks whose treewidths are too large for more traditional inference algorithms [Dar03, CD08, SCD16, PZ03].

Traditional representations of context-specific independence (CSI) use data structures such as decision trees, decision graphs, rules, default tables, etc. [FG98, CHM13, LD03, KF09, PNK15]. Algorithms for learning these representations are typically search-based, where we iteratively search for variables that split the data into partitions, until the resulting distribution becomes (sufficiently) independent of the remaining variables. In contrast to these heuristic methods, exact learning algorithms have also been proposed [KS05, HPK18]. Their running times are usually exponential in the number of parent variables, and it is difficult to apply these exact learning algorithms to a large system.

In this chapter, we propose a new perspective on learning CSIs, resulting in a new context-specific representation for conditional probability distribution (CPDs) that we call Functional Context-Specific CPDs, or just FoCS CPDs. FoCS CPDs generalize rule CPDs, where a context is typically defined as a partial instantiation of the variables. More recently, the Conditional Probabilistic Sentential Decision Diagram (or Conditional PSDD) was proposed, and further generalizes term-based rules to arbitrary propositional sentences [SCD18]. FoCS CPDs generalize this further so that an arbitrary function can be used to define the scope of a context, say one defined by a neural network.

The first significance of this new representation is that it allows us to immediately leverage powerful machine learning systems that have been developed in recent years, for the purposes of learning CSIs. The second significance is that efficient probabilistic reasoning can be enabled, by exploiting recently developed analytic tools from the domain of eXplainable Artificial Intelligence (XAI),[1] which allows us to extract a decision graph representation of a context-specific CPD, but one that facilitates exact inference, i.e., a conditional PSDD [SCD18, SGD19, SCD16].

This chapter is organized as follows. In Section 7.2, we review functional and context-specific representations of CPDs. In Section 7.3 we propose the FoCS CPD. In Section 7.4 we propose an algorithm to learn FoCS CPDs from data, and in Section 7.5 we show how to reason with them. We empirically compare FoCS CPDs with functional and context-specific representations in Section 7.6, and we provide a case study on "learning to decode" in Section 7.7. Finally, we conclude in Section 7.8.

7.2 Representations of CPDs

A Bayesian network (BN) has two main components: (1) a directed acyclic graph (DAG) and (2) a set of conditional probability tables (CPDs) [Pea89, Dar09, KF09, Mur12]. Typically,

[1]`https://www.darpa.mil/program/explainable-artificial-intelligence`.

CPDs are represented using tabular data structures, although this becomes impractical when a variable has many parents. In this section, we review two alternative representations of interest: functional representations (such as noisy-or models and neural networks) and context-specific representations, such as tree CPDs and rule CPDs.

In what follows, we use upper case letters (X) to denote variables and lower case letters (x) to denote their values. Variable sets are denoted by bold-face upper case letters ($\mathbf{X}$) and their instantiations by bold-face lower case letters ($\mathbf{x}$). Generally, we use X to denote a variable in a Bayesian network and $\mathbf{U}$ to denote its parents. We further refer to $X\mathbf{U}$ as a family. We thus denote a network parameter using the form $\theta_{x|\mathbf{u}}$, which represents the conditional probability $Pr(X=x|\mathbf{U}=\mathbf{u})$.

7.2.1 Functional Representations

To specify a CPD using a table, one must specify a parameter $\theta_{x|\mathbf{u}}$ for all family instantiations $x\mathbf{u}$, the number of which is exponential in the number of variables in the family $X\mathbf{U}$. In a functional representation of a CPD, one has a parameterized function $f(X\mathbf{U};\theta)$ that *computes* the probability $Pr(x|\mathbf{u})$ from a parameter vector θ that can be much smaller than the size of an explicit table. For example, the well-known noisy-or model implicitly specifies a conditional distribution using a number of parameters that is only linear in the number of parents [Pea89, Dar09].

Other functional representations include logistic functions [Fre98, Vom06] as well as neural networks [BB00, KW14]. Consider the following conditional distribution for a variable X with parents U_1U_2, where each variable is binary (0/1): $Pr(x=1 \mid u_1u_2) = \sigma(\theta_0+\theta_1u_1+\theta_2u_2)$, where $\sigma(a) = [1 + \exp\{-a\}]^{-1}$ is the sigmoid function and where $\theta_0, \theta_1, \theta_2$ are parameters. This functional CPD has the following tabular representation:

u_1, u_2	$1,1$	$1,0$	$0,1$	$0,0$
$Pr(X=1 \mid u_1u_2)$	$\sigma(\theta_0+\theta_1+\theta_2)$	$\sigma(\theta_0+\theta_1)$	$\sigma(\theta_0+\theta_2)$	$\sigma(\theta_0)$
$Pr(X=0 \mid u_1u_2)$	$1-\sigma(\theta_0+\theta_1+\theta_2)$	$1-\sigma(\theta_0+\theta_1)$	$1-\sigma(\theta_0+\theta_2)$	$1-\sigma(\theta_0)$

In general, if we have n binary parents U, then the tabular representation will have 2^n free parameters, whereas the functional logistic representation will have only $n + 1$ parameters.

While functional representations allow us to compactly specify a CPD, they become unwieldly once we need to perform any reasoning. For example, the following result shows that computing the Most Probable Explanation (MPE) is intractable when using a logistic representation of a CPD, even when the parents are independent.

The proof follows by reduction from the knapsack problem. Each item corresponds to a parent variable, and $|\mathbf{U}|$ equals the number of items. The value of each item is translated to the marginal probability of each parent variable, and the weight of each item is translated to the coefficient of the parent variable in the logistic function. The optimal selection of the items can be obtained from the MPE solution over the parent variables given that the child equals 0; an item is selected if the corresponding parent variable is assigned to 1 in the MPE solution.

Theorem 11. *Consider a prior probability $Pr(\boldsymbol{U})$ over n variables and a conditional probability $Pr(X \mid \boldsymbol{U})$. If the prior probability is fully factorized and $Pr(X \mid \boldsymbol{U})$ is represented by a logistic function using $n + 1$ parameters, it is NP-complete to compute* $\text{argmax}_{\boldsymbol{u}} Pr(\boldsymbol{u} \mid x)$.

7.2.2 Context-Specific CPDs

A *decision-tree CPD*, or just *tree CPD*, represents a conditional distrubtion of a variable X given its parents $\mathbf{U}$ in a Bayesian network [FG98, dRG05]. It is composed of a decision tree over variables $\mathbf{U}$, and at each leaf of the decision tree is a CPD column, which we denote by $\Theta_{X|.}$. A *CPD column* $\Theta_{X|.}$ is a distribution over variable X for some given context. A *decision-graph CPD* is a representation like a decision tree, but where equivalent leaves with equivalent CPD columns $\Theta_{X|.}$ are merged together in a single node [CHM13]. This decision graph can be further simplified by iteratively merging decision nodes whose children are equivalent. A *rule CPD* is another representation of a conditional distribution that uses rules to define the CPD. A rule is composed of two parts: a context, which is typically a partial instantiation $\mathbf{v}$ of the parent variables $\mathbf{U}$, and a CPD column $\Theta_{X|.}$. For example, a labeled DAG (LDAG) was introduced to specify these partial instantiation contexts on each edge of a DAG [PNK15, HPK18]. A set of rules specify a rule CPD if the contexts $\mathbf{v}$ represent a mutually-exclusive and exhaustive partitioning of the instantiations of $\mathbf{U}$.

A decision-tree CPD specifies a set of rules, where each leaf represents a rule with the same CPD column $\Theta_{X|.}$ assigned to the leaf, where the context $\mathbf{v}$ is found by taking the value of each variable that was branched on, on the path from the root to the leaf. A decision-graph CPD also specifies a set of rules in a similar way, except that we relax the requirement that the context be specified by a partial instantiation, but now as a disjunction of partial instantiations, one for each path that can reach the leaf from the root. The rule CPD can be generalized further by allowing the context to be specified as an arbitrary propositional sentence. Such a rule CPD can be realized using the recently proposed Conditional Probabilistic Sentential Decision Diagram (Conditional PSDD) [SCD18].[2] While they enable more succinct representations of conditional distributions, Conditional PSDDs also facilitate the ability to reason with them [SCD16, SGD19].

7.3 Functional Context-Specific CPDs

Next, we propose a generalized rule CPD where the scope of a rule is defined, not just by a partial instantiation of its parents, or just by a propositional sentence, but more generally by some function.

Definition 17. *A Functional Context-Specific (FoCS) CPD represents a conditional distribution of a variable X given its parents $\boldsymbol{U}$, and is defined by a tuple $(f, I_{1\cdots k}, \Theta_{X|I_{1\cdots k}})$. The function f maps each parent configuration to a real number. Intervals I_i form a mutually-exclusive and exhaustive partition of $\mathbb{R}$. Each interval, I_i, has a corresponding child probability that is specified by $\Theta_{X|I_i}$.*

A FoCS CPD, which we denote by $\Phi_{X|\mathbf{U}}$, induces a conditional distribution $Pr(X|\mathbf{U})$

[2]In a Conditional PSDD, the contexts are represented using a shared SDD [Dar11], and the CPD columns are represented using a shared PSDD [KVC14].

where:

$$Pr(X \mid \mathbf{u}) = \begin{cases} \Theta_{X|I_1} & \text{if } f(\mathbf{u}) \in I_1 \\ ... & ... \\ \Theta_{X|I_k} & \text{if } f(\mathbf{u}) \in I_k \end{cases}. \tag{7.1}$$

Each interval I_i defines a context Δ_i. The models of the context are the parent configurations whose function values fall inside I_i. Since the intervals are mutually exclusive and exhaustive, the contexts form a partition of parent instantiations, and the conditional distribution is well-defined.

Further, the k intervals of a FoCS CPD using $k-1$ monotonically increasing thresholds, $T_1, \cdots, T_{k-1}$. These thresholds create intervals, $(-\infty, T_1), [T_1, T_2), \cdots, [T_{k-1}, +\infty)$. As a result, we will interchangeably use the notation $I_{1\cdots k}$ and $T_{1\cdots k-1}$ to describes the intervals.

7.4 Learning

In this section, we show how to (1) learn the parameters of a FoCS CPD when the contexts are known, and (2) how to learn the contexts of a FoCS CPD, using parameter learning as a sub-routine.

7.4.1 Learning the Parameters

Given a dataset $\mathcal{D}$, the log likelihood of a set of Bayesian network parameters Θ is

$$LL(\mathcal{D} \mid \Theta) = \textstyle\sum_{X\mathbf{U}} CLL(\mathcal{D}_{X\mathbf{U}} \mid \Theta_{X|\mathbf{U}})$$

which is the sum of the conditional log likelihoods of the CPDs $\Theta_{X|\mathbf{U}}$ given the datasets $\mathcal{D}_{\mathbf{XU}}$ projected onto the families $X\mathbf{U}$. The local conditional log likelihoods is given by

$$CLL(\mathcal{D}_{X\mathbf{U}} \mid \Theta_{X|\mathbf{U}}) = \textstyle\sum_{i=1}^{N} \log \theta_{x_i|\mathbf{u}_i}$$

which we can optimize independently. Let $\mathcal{D}\#(\mathbf{y})$ denote the number of instances in the dataset $\mathcal{D}$ compatible with the partial instantiation $\mathbf{y}$. The parameters $\theta^{\star}_{x|\mathbf{u}}$ that optimize the conditional log likelihood is given by $\theta^{\star}_{x|\mathbf{u}} = \frac{\mathcal{D}\#(x\mathbf{u})}{\mathcal{D}\#(\mathbf{u})}$ which further represent the maximum-likelihood estimates; for more details, see, e.g., [Dar09, KF09, Mur12].

Analogously, estimates can be obtained for a Bayesian network with FoCS CPDs $\Phi_{X|\mathbf{U}}$ with contexts Δ_i and the corresponding CPD columns $\Theta_{X|\Delta_i}$. Namely, we have the maximum-likelihood estimates $\theta^{\star}_{x|\Delta_i} = \frac{\mathcal{D}\#(x,\Delta_i)}{\mathcal{D}\#(\Delta_i)}$. That is, we count the number of instances compatible with both x and the context Δ_i, and then normalize by the total number of instances compatible with the context Δ_i.

Each maximum likelihood estimates $\theta^{\star}_{x|\Delta_i}$ for a FoCS CPD $\Phi_{X|\mathbf{U}}$ can be computed using a single pass of the dataset $\mathcal{D}$, and also proportional to the time it takes to test whether a given example $\mathbf{d}$ from the dataset is compatible with Δ_i. As contexts of a FoCS CPD are specified by a function and a set of intervals $I_{1\cdots k}$, it suffices to evaluate the function at the given partial instantiation $\mathbf{u}$ and then test whether the function value lies in the corresponding internal I_i.

7.4.2 Learning the Contexts

Next, we propose a simple algorithm to learn the contexts of a FoCS CPD from a given dataset $\mathcal{D}$. Based on the definition of a FoCS CPD, its context is defined by a function, that maps parent configurations to a number, and a set of intervals described by thresholds $T_{1\cdots k-1}$. The context Δ_i consists of parent configurations whose function value fall inside the interval I_i.

Our approach has two steps: (1) we first construct the function by learning a functional CPD using an MLP, like the CPDs we discussed in Section 7.2.1, and then (2) we iteratively learn thresholds on the output of the MLP. As we showed in Theorem 11, MPE inference using a functional representation of a CPD is in general intractable, like the one we shall learn in Step (1). As we shall discuss later in Section 7.5, the FoCS CPDs that we obtain from Step (2) shall give us a way to approach this apparently intractability.

First, we learn a functional representation of the conditional distribution $Pr(X \mid \mathbf{U})$. Here, we use an MLP, which we denote by $f_x(\mathbf{u})$, to estimate the conditional probabilities

	U_1	U_2	X
$\mathbf{d}_1$	0	0	1
$\mathbf{d}_2$	1	0	0
$\mathbf{d}_3$	1	1	0
$\mathbf{d}_4$	0	1	1
$\mathbf{d}_5$	1	1	1

(a)

	U_1	U_2	f
$\mathbf{d}_2$	1	0	$\sigma(-2)$
$\mathbf{d}_4$	0	1	$\sigma(-1)$
$\mathbf{d}_3$	1	1	$\sigma(1)$
$\mathbf{d}_5$	1	1	$\sigma(1)$
$\mathbf{d}_1$	0	0	$\sigma(2)$

(b)

T	$\Delta_{\leq T}$	$\Delta_{>T}$
$-\infty$	$\{\}$	$\{\mathbf{d}_2, \mathbf{d}_4, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_1\}$
$\sigma(-2)$	$\{\mathbf{d}_2\}$	$\{\mathbf{d}_4, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_1\}$
$\sigma(-1)$	$\{\mathbf{d}_2, \mathbf{d}_4\}$	$\{\mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_1\}$
$\sigma(1)$	$\{\mathbf{d}_2, \mathbf{d}_4, \mathbf{d}_3, \mathbf{d}_5\}$	$\{\mathbf{d}_1\}$
$\sigma(2)$	$\{\mathbf{d}_2, \mathbf{d}_4, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_1\}$	$\{\}$

(c)

Figure 7.1: (a) a dataset, (b) sorted by MLP output, (c) different thresholds and the resulting partitions.

$Pr(x \mid \mathbf{u})$ for some distinguished state x of a variable X; for simplicity, we assume X is binary (0/1). In particular, the MLP is trained using feature-label pairs $(\mathbf{u}, x)$ for all family instantiations $x, \mathbf{u}$ that appear in the original dataset $\mathcal{D}$. In our experiments, we used cross entropy as a loss function.

Our next step is to obtain a FoCS CPD $\Phi_{X|\mathbf{U}}$ from the MLP $f_x(\mathbf{U})$ that we have just learned. Our approach is based on learning a threshold on the output of our MLP, which in turn induces a partition of the input space. By iteratively learning additional thresholds, we can further refine our partitioning. Suppose for now that we learn a single threshold T, which yields the following contexts: $\Delta_{\leq T} = \{\mathbf{u} \mid f_x(\mathbf{u}) \leq T\}$ and $\Delta_{>T} = \{\mathbf{u} \mid f_x(\mathbf{u}) > T\}$ where $\Delta_{\leq T} = \neg\Delta_{>T}$ relative to all parent instantiations $\mathbf{u}$, i.e., we have a partitioning.

Consider Figure 7.1, which highlights a simple example. In Figure 7.1a, we have a small dataset. Suppose that we learn the following MLP f from this dataset: $f(u_1, u_2) = \sigma(6u_1u_2 - 4u_1 - 3u_2 + 2)$. In Figure 7.1b, we have sorted this dataset by the value of $f(\mathbf{u})$; remember that the sigmoid function σ is a monotonic non-decreasing function, i.e., $\sigma(x) \leq \sigma(y)$ iff $x \leq y$. Note that for any two consecutive output values f_i and f_j in the sorted list of Figure 7.1b, any chosen threshold $T \in (f_i, f_j]$ will result in the same partitioning. For example a threshold $T = \sigma(0)$ results in the same partition as threshold $T = \sigma(0.9)$, yielding the sets $\Delta_{\leq T} = \{\mathbf{d}_2, \mathbf{d}_4\}$ and $\Delta_{>T} = \{\mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_1\}$ (note that $\mathbf{d}_3$ and $\mathbf{d}_5$ represent the same parent instantiation $u_1 = 1, u_2 = 1$).

For each threshold T, we can learn the resulting parameters $\theta_{X|\Delta_{\leq T}}$ and $\theta_{X|\Delta_{>T}}$ using a single pass over the dataset, and then compute the resulting conditional log likelihood. If N is the size of the dataset $\mathcal{D}$, then it suffices to check N possible threshold values, plus one additional threshold $T = -\infty$ that ensures that $\Delta_{\leq T}$ is empty, and that $\Delta_{>T}$ contains all of the examples. We then simply pick the single threshold that maximizes the conditional log likelihood. Finally, one can amortize the complexity of computing the conditional log likelihoods for all possible thresholds, hence requiring a single pass over the dataset $\mathcal{D}$ overall.

We can refine the partition further by recursing on each partition, and finding an additional threshold within each partition, using the same algorithm we described above. We can continue to recurse and refine our partition, until validation likelihood falls or does not improve enough.

7.5 Reasoning

In general, if we use a purely functional representation of a CPD, then inference becomes intractable, as given by Theorem 11. Alternatively, we seek next to obtain tractable FoCS CPDs, first using recently proposed analytic tools from the domain of explainable AI (XAI).

7.5.1 Marginal Inference via Knowledge Compilation

Recently, in the domain of XAI, [CSS19] showed how a binary neural network (BNN) can be formally analyzed and verified using symbolic tools from the domain of Knowledge Compilation [DM02]. A BNN is a neural network with binary inputs and a binary output. Such a neural network represents a Boolean function. Consider for example a linear classifier f: $1.15 \cdot U_1 + 0.95 \cdot U_2 - 1.05 \cdot U_3 \geq 0.52$. Here, U_1, U_2, U_3 are binary (0/1) inputs, and the classifier outputs 1 if this threshold test passes and it outputs 0 otherwise. We can enumerate all possible inputs and record the classifier output $f(u_1, u_2, u_3)$, leading to the following truth table:

U_1	U_2	U_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0

U_1	U_2	U_3	f
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

The original numerical linear classifier is thus equivalent to the propositional sentence $[\neg U_3 \wedge (U_1 \vee U_2)] \vee [U_3 \wedge U_1 \wedge U_2]$. Previously, [CD03] showed how to extract the Boolean function of a given linear classifier, which includes neurons with step activations as a special case. More recently, [CSS19] showed that one can compose the Boolean functions of binary neurons and aggregate them to obtain the Boolean function of a binary neural network. By compiling this Boolean function into a tractable logical representation, such as an Ordered Binary Decision Diagram (OBDD) or as a Sentential Decision Diagram (SDD), then certain queries and transformations can be performed in time that is polynomial in the size of the resulting circuit [DM02, Dar11].

Here, we use the algorithm proposed by [CSS19], to compile a binary neural network into an SDD circuit, in order to compile a FoCS CPD into a Conditional PSDD. First, if we threshold the output of an MLP with step-activations, then it corresponds to a binary neural network. Hence, we can compile each FoCS CPD context Δ_i into an SDD. Second, it is straightforward to compile a CPD column $\Theta_{X|\cdot}$ into a PSDD [SCD18]. It is then straightforward to aggregate all of the context SDDs and CPD column PSDDs into a single Conditional PSDD [SCD18]. If we obtain the CPDs of a Bayesian network as a Conditional PSDD, then we can employ the algorithms in [SGD19, SCD16], in order to compute marginals in the Bayesian network.[3]

7.5.2 MPE Inference via Mixed-Integer Linear Programming

Consider the most probable explanation (MPE) query in a Bayesian network: $\operatorname{argmax}_{\mathbf{x} \sim \mathbf{e}} Pr(\mathbf{x})$, where $\mathbf{x}$ is a complete instantiation of the network variables, $\mathbf{e}$ is the observed evidence, and

[3]Note that while multiplying two PSDDs is a tractable operation, multiplying n PSDDs may not be.

$\sim$ denotes compatiability between $\mathbf{x}$ and $\mathbf{e}$ (they set common variables to the same values). Computing the MPE is an NP-complete problem [Shi94]. Theorem 11 shows that MPE is still NP-complete with independent parents $\mathbf{U}$ and a common observed child X with a functional CPD. However, MPE is still easier than computing marginals, which is PP-complete to even approximate [Rot96]. Hence, compiling to conditional PSDD may be overkill if we only care about MPEs.

Given a FoCS CPD, it suffices to apply a mixed-integer linear programming (MILP) solver to the task of solving an MPE query, i.e., to compute $\text{argmax}_{\mathbf{u}}\, Pr(\mathbf{u}, x)$ where $Pr(X|\mathbf{U})$ is represented with a FoCS CPD. First, the log of the MPE is a linear function of the log parameters of the network parameters, which we use as the objective of the MILP. Using a FoCS CPD for a binary variable X, an observation x effectively adds another term to the objective function, which depends on the context implied by the input $\mathbf{u}$. This can be incorporated to the MILP after observing that an MLP with step activations can be reduced to an MILP, as in [NKR18, GNS08].

7.6 Experiments

In this section, we empirically evaluate the FoCS CPD representation that we proposed in Section 7.3, as well as the learning algorithm that we proposed in Section 7.4. In particular, we evaluate it in terms of our effectiveness at learning conditional distributions, in comparison to other functional and context-specific representations. We shall subsequently evaluate the reasoning algorithms proposed in Section 7.5, via a case study in Section 7.7.

We evaluate two sets of benchmarks, one synthetic, and one real-world. We consider two baselines: (1) a functional CPD representation, using a multi-layer perceptron (MLP), and (2) a context-specific CPD representation, namely a tree CPD, which was learned using ID3. We compare each representation based on their (negated) conditional log likelihood (CLL); lower is better. When learning any CPD column, we further use Laplace (add-one) smoothing.

We trained an MLP $f_x(\mathbf{U})$ to predict the value of variable X given an instantiation

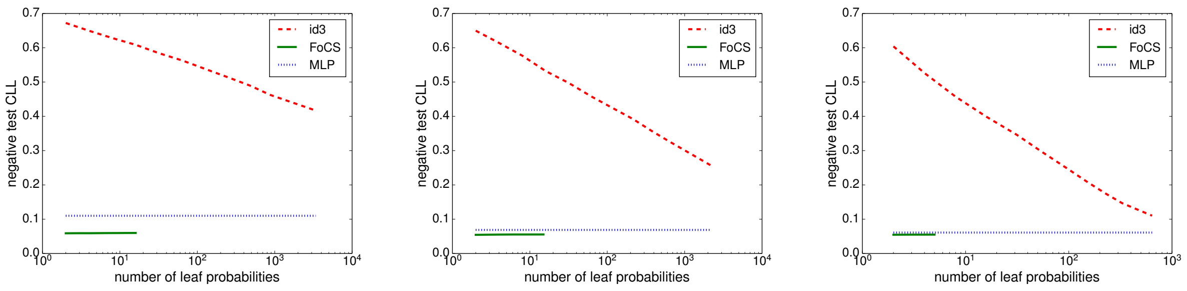


Figure 7.2: Number of contexts vs. CLL, for $k \in \{2, 4, 8\}$ from left to right (synthetic benchmark).

of the parents $\mathbf{U}$. We used a single hidden layer of 16 neurons with ReLU activations, whose parameters were learned with cross-entropy loss. The MLP was trained using the Adam optimizer in TensorFlow. We trained a tree CPD using the ID3 algorithm of the toolkit scikit-learn. In our experiments, we learned decision trees of gradually increasing complexity (measured by counting decision tree leaves), by gradually increasing the bound on tree depth, which is a parameter of the ID3 algorithm.

Finally, to obtain a FoCS CPD, we use the learning algorithm described in Section 7.4, using the MLP that we trained above. In our experiments, we also gradually increased the number of contexts created. With $k-1$ thresholds created, we create k contexts, which we compare with the number of contexts found by the tree CPD (i.e., the number of decision tree leaves).

Synthetic Benchmark. In our synthetic experiment, we simulated training data from a conditional distribution exhibiting many context-specific independencies. That is, we simulated data where the value of variable X depends only on the *cardinality* of its parents $\mathbf{U}$, where X and $\mathbf{U}$ are binary (0/1). If the fraction of parents set to 1 is at most $\frac{1}{s}$, then $Pr(x|\mathbf{u}) = 0.05$; otherwise $Pr(x|\mathbf{u}) = 0.95$. In our experiments, we simulated parent instantations $\mathbf{u}$ such that the two different contexts ($\leq \frac{1}{s}$ and $> \frac{1}{s}$) had the same probability of being generated. The size of parent variables was set to be 16. Each FoCS CPD was trained using $16,384$ examples, and we reported the negative CLL on $16,384$ testing examples.

In Figure 7.2, we plot results for $s \in \{2, 4, 8\}$. On the x-axis we increase the number of contexts for the tree CPD (by increasing the bound on depth) and for the FoCS CPD (by adding more thresholds); the MLP is a functional CPD without any explicit CSIs, and hence is a flat line on each plot. We make the following observations in Figure 7.2: (1) as we increase the number of contexts of the tree CPD (ID3), the better the CLL, (2) the MLP and the FoCS CPD perform similarly, and both obtain better CLLs than the tree CPD, and (3) the FoCS CPD obtains a good CLL using only a small number of contexts, and obtains a better CLL than the MLP that it was created from.

It is well-known that decision trees cannot succinctly represent certain (Boolean) functions. For example, a decision tree must be complete, using 2^n leaves, to represent the parity function over n variables. This is also the case for cardinality constraints over n variables, which have less succinct decision trees for $s = 2$ and succinct decision trees for $s = 1$ or $s = n$. We see this pattern as well in Figure 7.2, as the performance of ID3 more closely approaches that of MLP and our FoCS CPD.

Compared to the MLP, our FoCS model estimates much fewer parameters—once we are given k contexts Δ_i, then we simply need to estimate the k corresponding CPD columns $\Theta_{X|\Delta_i}$. The fact that our learning appears to converge almost immediately, suggests that our learning algorithm is indeed learning the context-sensitive inherent in the cardinality constrained data that we simulated. In contrast, the MLP does not search for CSIs. Hence, this explains the ability of our FoCS model to obtain better CLLs than the MLP that our FoCS model was based on.

Real-World Benchmark. Next, we consider a real-world dataset: MNIST digits. This dataset is composed of 28×28 pixel grayscale images, which we binarized to black-and-white. We consider one-vs-all classification, where the parents $\mathbf{U}$ represent the input image, and the child X represents whether the input is of a particular digit d ($X = \mathsf{true}$) or some other digit from 0 to 9 ($X = \mathsf{false}$).

Figure 7.3 highlights the result. Our FoCS model consistently estimates the conditional distribution more accurately using fewer contexts compared to the tree CPD model. This

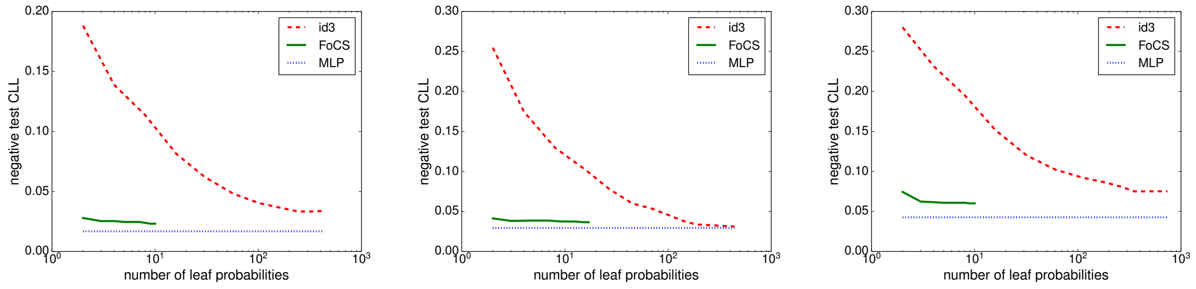


Figure 7.3: Number of contexts vs. CLL, for digits $d \in \{0, 1, 2\}$ from left-to-right (MNIST).

provides strong evidence that exploiting the structure from a learned functional model is more efficient than searching for the structure of contexts by variable-splitting as done when learning a decision tree. Again, given that our FoCS model appears to converge relatively quickly, this suggests that our learning algorithm is able to learn the CSIs from data (a CSI may represent here a partial instantiation of the input pixels that will almost guarantee the classification of a digit). However, a small number of contexts may not be enough to obtain the performance of an MLP. Note that there may be only a limited number of contexts that our learning algorithm can discover, if the training data is not diverse enough, hence why the curve for our FoCS model stops earlier than that for the tree CPD.[4]

7.7 Case Study: Learning to Decode

In this section, we show through a simple case study how FoCS CPDs allow us to reason about and learn from complex processes. Our case study is done in the context of channel coding, where our goal is to encode data in a way that allows us to detect and correct for any errors that may occur after transmission through a noisy channel. In subsequent experiments, we show, using our proposed learning algorithm, how one can learn to decode encoded messages, without knowing the original code that was used to encode them!

[4]For example, if the MLP is trained to the point where it obtains 100% confidence in most of the training examples (which is not unlikely for datasets such as MNIST), then we would not be able to split any of the resulting contexts).

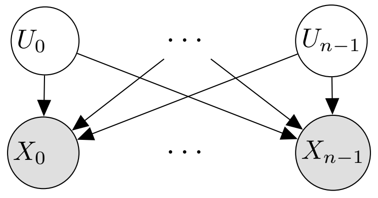


Figure 7.4: The Bayesian network modeling the encoding process.

7.7.1 Channel Coding: A Brief Introduction

Consider the following problem. Say you have a message represented using n bits $U_0, \ldots, U_{n-1}$ that can be either 0 or 1. We want to transmit this message across a noisy channel, where there is a chance that each bit U_i might be corrupted by noise (say flipped from 0 to 1, or from 1 to 0). To improve the reliability of this process we can send additional bits, say m of them $X_0, \ldots, X_{m-1}$. We refer to the original bits $\mathbf{u}$ as the *message* and the redundant bits $\mathbf{x}$ as the *encoding* (or alternatively the *channel input*). We further refer to the encoding process as the *code*. The *channel output* are the bits $\mathbf{y}$ received from the noisy channel. Finally, there is a *decoding* process that attempts to detect and correct any errors in the channel output.

One simple example of a code, is the repetition code, which sends l additional copies of the message across the noisy channel (say 3 copies total). At the channel output, one detects an error if any of the 3 copies reports a discrepency among the corresponding bits. One can attempt to correct for the error by taking a majority vote. Repetition codes are among the simplest type of error-correcting code. More sophisticated codes include *turbo codes* and *low density parity check codes*, whose decoders were shown to be instances of loopy belief propagation in a Bayesian network; see [FM97] for a short perspective. Another common type of code uses parity checks among randomly selected sets of message bits, as redundant bits in the encoding.

7.7.2 Experiments

We can model a message encoder using a Bayesian network like the one in Figure 7.4. Root nodes U_i represent the bits (0/1) to be encoded, and the leaf nodes X_i represent the encoded bits (0/1) that are to be transmitted across a noisy channel. For simplicity, we assume that both the message and the encoding are composed of n bits. We assume the message bits U_i are marginally independent. Each of the encoded bits X_i can in general depend on any of the message bits U_i, depending on the particular code being used. At the same time, we can model the noisy channel, which may flip a bit from 0 to 1 or from 1 to 0, with some probability. The traditional reasoning task would be, given a code, i.e., the conditional distributions $Pr(X_i \mid \mathbf{U})$, and a message $\mathbf{x}$ received over the noisy channel, to find the most likely message $\mathbf{u}$ that was originally encoded.

Consider for example a simple code where we record the parity of every pair of adjacent message bits. Such an encoding, can be represented using the following CPD:

$$Pr(X_i{=}1 \mid \mathbf{U}) = \begin{cases} 100\% & \text{if } U_i \oplus U_{i+1 \bmod n} = 1 \\ 0\% & \text{if } U_i \oplus U_{i+1 \bmod n} = 0 \end{cases}.$$

We can further model the noise in the channel with the following modified CPD.

$$Pr(X_i{=}1 \mid \mathbf{U}) = \begin{cases} 95\% & \text{if } U_i \oplus U_{i+1 \bmod n} = 1 \\ 5\% & \text{if } U_i \oplus U_{i+1 \bmod n} = 0 \end{cases}.$$

Given a set of message/encoding pairs $(\mathbf{u}, \mathbf{x})$ we can try to learn the code used to encode the messages, i.e., learn the conditional distributions $Pr(X_i \mid \mathbf{U})$. We represent each conditional distribution using a FoCS CPD, as a tabular representation would be intractable for this type of problem: the table would have a number of entries that is exponential in n, and we would need at least as much data to learn the parameters.

We learn a FoCS CPD as described in Section 7.4, starting with an MLP with a single hidden layer containing 8 neurons with sigmoid activations. We convert the sigmoid activations to step activations for the purposes of reducing it to MILP, in order to perform MPE inference as described in Section 7.5. From the MLP, we obtain a FoCS CPD by learning one threshold, which yields 2 different contexts.

Once we have learned a FoCS model from data, we can then try to decode an encoded message. That is, given an encoded message $\mathbf{x}$, we can find the most likely original encoding via: $\text{argmax}_{\mathbf{u}} \, Pr(\mathbf{u} \mid \mathbf{x})$, which is an MPE query. We used the MILP solver Gurobi [Gur20] with the cvxpy optimizer.

We want to recognize that this task also known as a structured prediction. Many existing methods can directly train a model for this particular MPE queries, [TJH05, SM12, SMV19, XZF18]. However, learning a query-specific model prevents us from taking advantage of the generation process of these bits, which is shown in Figure 7.4.

To obtain a training and testing set of messages $\mathbf{u}$ we sampled bits at random with $Pr(u_i) = 0.8$. To obtain a set of encoded messages $\mathbf{x}$, we used a code where each encoded bit took the parity of three consecutive bits (there are n such encoded bits). We assume that the channel has a 5% chance of flipping an transmitted bit. We simulated datasets of size $2^{14} = 16,384$, and performed 5-fold cross validation. The following table summarizes our results.

n	word accuracy	bit accuracy	Hamming error	time (s)
10	0.750 ± 0.003	0.902 ± 0.002	0.974 ± 0.021	0.247 ± 0.001
15	0.651 ± 0.005	0.900 ± 0.002	1.493 ± 0.044	0.469 ± 0.004
20	0.578 ± 0.005	0.905 ± 0.001	1.886 ± 0.037	1.047 ± 0.036
25	0.493 ± 0.007	0.905 ± 0.001	2.371 ± 0.043	11.382 ± 0.549
30	0.414 ± 0.006	0.901 ± 0.003	2.963 ± 0.099	140.190 ± 11.539

From top-to-bottom, each row represents increasing message sizes. We report word accuracy (the percentage of instances where the original message was successfuly decoded from the encoding without error), bit accuracy (the percentage of bits that were decoded without error), Hamming error (the average number of incorrect bits in a decoded message), and time (in seconds).

We make a few observations. Bit accuracy remains consistent around 90%, for all message sizes n. Word accuracy falls, as expected, since it becomes more difficult to decode the entire message without error, the longer the message gets. Note that a 41.4% for $n = 30$ is quite good compared to the expected word accuracy one would have obtained by composing a message estimate from most-likely-bit estimates at 90% accuracy, which would be $0.9^{30} = 4.24\%$. When we consider the hamming error, even if there were an error in the decoding, only a few bits were incorrect on average. Finally, we see that inference time appears to grow exponentially as n grows. This is also expected as decoding is in general an NP-hard problem.

7.8 Conclusion

We proposed here the FoCS CPD model, for representing CSIs in conditional distributions. We proposed an algorithm for learning the parameters as well as the contexts of FoCS CPDs. We showed how efficient inference can be enabled using FoCS CPDs, by leveraging tools from knowledge compilation and optimization. We highlighted some of the advantages of FoCS CPDs compared to more traditional functional and context-sensitive CPD representations. Finally, we provided a case study showing how FoCS CPDs enable us to "learn how to decode."

CHAPTER 8

Conclusion

We considered two types of knowledge in this dissertation: global conditional independence and local context-specific independence. We tackled the problem of incorporating these two types of knowledge in a unified framework. Traditionally, BNs achieved this goal to some extent; it uses a DAG to capture the global structure, and local context-specific independence is modeled using a structured representations of conditional probabilities, e.g. tree CPT. However, this was not sufficient when massive logical constraints were present in the problem; the DAG of the BN became so densely connected that problem structure was hidden. Furthermore, modeling local structure at the variable level missed opportunities to extract higher-order patterns that were implied by logical constraints.

To address these limitations, we proposed structured Bayesian networks. In contrast to regular BNs, SBNs used a cluster DAG to represent global knowledge. The cluster DAG produced a weaker global structure, by keeping silent about the structure among variables in the same cluster. We further proposed and developed conditional PSDDs to capture the local structure between a cluster and its parents. As the conditional probabilities of an SBN describes dependencies between *sets* of variables, conditional PSDDs captured higher-order structures, especially with conditional constraints.

We later studied the inference problem on SBNs. We first studied the tractable operations on (conditional) PSDDs, which were used to model the local probabilities of an SBN. We have presented a simple inference algorithm that compiles an SBN to a single PSDD representing the same joint probability. The compilation utilized the tractable operation of PSDDs, called multiply.

On the application side, we thoroughly case studied the usage of this new framework on modeling a distribution over routes. We proposed hierarchical assumptions, from which we crafted a global structure, a cluster DAG, of the SBN. Each node in the cluster DAG was also logically connected with its parents, and this logical constrain formed the structure of the conditional PSDD that was used to model the probabilistic interactions between the node and its parents. Moreover, we presented a sub-class of the model that scales to the size of a city, and we have empirically shown that this model was good at many tasks, including route completion and classification.

At last, we considered the problem of learning conditional PSDDs solely from data. In particular, we demonstrated that context-specific independence can be extracted from a functional model. We have shown an algorithm that extracted this local structure from a trained functional model. The extracted local knowledge was then converted to conditional PSDDs, which was a more concrete conditional representation.

Overall, we hope that this dissertation can shine a light on the opportunities of incorporating more types of domain knowledge into a graphical model.

REFERENCES

[Bar12] David Barber. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 2012.

[BB00] S. Bengio and Y. Bengio. "Taking on the Curse of Dimensionality in Joint Distributions Using Neural Networks." *Trans. Neur. Netw.*, **11**(3):550–557, May 2000.

[BDC15] Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. "Tractable Learning for Complex Probability Queries." In *NIPS*, 2015.

[BFG96] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. "Context-Specific Independence in Bayesian Networks." In *UAI*, pp. 115–123, 1996.

[Bov16] Simone Bova. "SDDs Are Exponentially More Succinct than OBDDs." In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 929–935, 2016.

[CD03] Hei Chan and Adnan Darwiche. "Reasoning about Bayesian Network Classifiers." In *UAI*, pp. 107–115, 2003.

[CD06] Hei Chan and Adnan Darwiche. "On the Robustness of Most Probable Explanations." In *UAI*, 2006.

[CD07] Mark Chavira and Adnan Darwiche. "Compiling Bayesian Networks Using Variable Elimination." In *IJCAI*, 2007.

[CD08] Mark Chavira and Adnan Darwiche. "On Probabilistic Inference by Weighted Model Counting." *AIJ*, **172**(6–7):772–799, April 2008.

[CD11] Arthur Choi and Adnan Darwiche. "Relax, Compensate and Then Recover." In Takashi Onada, Daisuke Bekki, and Eric McCready, editors, *NFAI*, volume 6797 of *LNCF*, pp. 167–180. Springer, 2011.

[CD13] Arthur Choi and Adnan Darwiche. "Dynamic Minimization of Sentential Decision Diagrams." In *Proceedings of the 27th Conference on Artificial Intelligence (AAAI)*, 2013.

[CD17] Arthur Choi and Adnan Darwiche. "On Relaxing Determinism in Arithmetic Circuits." In *Proceedings of the Thirty-Fourth International Conference on Machine Learning (ICML)*, 2017.

[CHM13] David Maxwell Chickering, David Heckerman, and Christopher Meek. "A Bayesian Approach to Learning Bayesian Networks with Local Structure." *CoRR*, **abs/1302.1528**, 2013.

[CKD13] Arthur Choi, Doga Kisa, and Adnan Darwiche. "Compiling Probabilistic Graphical Models using Sentential Decision Diagrams." In *ECSQARU*, pp. 121–132, 2013.

[Coo90] Gregory F. Cooper. "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks (Research Note)." *Artif. Intell.*, **42**(2–3):393–405, March 1990.

[CSD17] Arthur Choi, Yujia Shen, and Adnan Darwiche. "Tractability in Structured Probability Spaces." In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 3477–3485, 2017.

[CSS19] Arthur Choi, Weijia Shi, Andy Shih, and Adnan Darwiche. "Compiling Neural Networks into Tractable Boolean Circuits." In *AAAI Spring Symposium on Verification of Neural Networks (VNN)*, 2019.

[CTD16] Arthur Choi, Nazgol Tavabi, and Adnan Darwiche. "Structured Features in Naive Bayes Classification." In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

[CVD15] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. "Tractable Learning for Structured Probability Spaces: A Case Study in Learning Preference Distributions." In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[Dar01a] Adnan Darwiche. "Decomposable Negation Normal Form." *J. ACM*, **48**(4):608–647, 2001.

[Dar01b] Adnan Darwiche. "On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision." *Journal of Applied Non-Classical Logics*, **11**(1-2):11–34, 2001.

[Dar03] Adnan Darwiche. "A Differential Approach to Inference in Bayesian Networks." *J. ACM*, **50**(3):280–305, 2003.

[Dar09] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press, USA, 1st edition, 2009.

[Dar11] Adnan Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases." In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pp. 819–826, 2011.

[DDC08] Adnan Darwiche, Rina Dechter, Arthur Choi, Vibhav Gogate, and Lars Otten. "Results from the Probabilistic Inference Evaluation of UAI-08." `http://graphmod.ics.uci.edu/uai08/Evaluation/Report`, 2008.

[DM02] Adnan Darwiche and Pierre Marquis. "A knowledge compilation map." *JAIR*, **17**:229–264, 2002.

[dRG05] Marie desJardins, Priyang Rathod, and Lise Getoor. "Bayesian Network Learning with Abstraction Hierarchies and Context-Specific Independence." In *ECML*, pp. 485–496, 2005.

[DV15] Aaron W. Dennis and Dan Ventura. "Greedy Structure Search for Sum-Product Networks." In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 932–938, 2015.

[FG98] Nir Friedman and Moises Goldszmidt. "Learning Bayesian networks with local structure." In *Learning in graphical models*, pp. 421–459. Springer, 1998.

[FM97] Brendan J. Frey and David J. C. MacKay. "A Revolution: Belief Propagation in Graphs with Cycles." In *NIPS*, pp. 479–485, 1997.

[Fre98] Brendan J. Frey. *Graphical Models for Machine Learning and Digital Communication.* MIT Press, Cambridge, MA, USA, 1998.

[GD12] Robert Gens and Pedro M. Domingos. "Discriminative Learning of Sum-Product Networks." In *Advances in Neural Information Processing Systems 25 (NIPS)*, pp. 3248–3256, 2012.

[GD13] Robert Gens and Pedro M. Domingos. "Learning the Structure of Sum-Product Networks." In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pp. 873–880, 2013.

[GNG14] William Groves, Ernesto Nunes, and Maria L. Gini. "A framework for predicting trajectories using global and local information." In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pp. 1–10, 2014.

[GNS08] Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization (2. ed.).* SIAM, 2008.

[Gur20] LLC Gurobi Optimization. "Gurobi Optimizer Reference Manual.", 2020.

[HKG12] Jonathan Huang, Ashish Kapoor, and Carlos Guestrin. "Riffled Independence for Efficient Inference with Partial Rankings." *J. Artif. Intell. Res. (JAIR)*, **44**:491–532, 2012.

[HPK18] Antti Hyttinen, Johan Pensar, Juha Kontinen, and Jukka Corander. "Structure learning for Bayesian networks over labeled DAGs." In *International Conference on Probabilistic Graphical Models*, pp. 133–144, 2018.

[Kam03] Toshihiro Kamishima. "Nantonac collaborative filtering: recommendation based on order responses." In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 583–588, 2003.

[KF09] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, 2009.

[Kru08] John Krumm. "A Markov model for driver turn prediction." Technical report, SAE Technical Paper, 2008.

[KS05] Mikko Koivisto and Kismat Sood. "Computational aspects of Bayesian partition models." In *Proceedings of the 22nd international conference on Machine learning*, pp. 433–440, 2005.

[KVC14] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. "Probabilistic Sentential Decision Diagrams." In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2014.

[KVD17] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. "Algebraic model counting." *J. Applied Logic*, **22**:46–62, 2017.

[KW14] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes." In *ICLR*, 2014.

[LB11] Tyler Lu and Craig Boutilier. "Learning Mallows Models with Pairwise Preferences." In *ICML*, pp. 145–152, 2011.

[LBB17] Yitao Liang, Jessa Bekker, and Guy Van den Broeck. "Learning the Structure of Probabilistic Sentential Decision Diagrams." In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

[LD03] David Larkin and Rina Dechter. "Bayesian Inference in the Presence of Determinism." In *AISTATS*, 2003.

[LD08] Daniel Lowd and Pedro M. Domingos. "Learning Arithmetic Circuits." In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI)*, pp. 383–392, 2008.

[LR13] Daniel Lowd and Amirmohammad Rooshenas. "Learning Markov Networks With Arithmetic Circuits." In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 406–414, 2013.

[Mal57] Colin L. Mallows. "Non-null ranking models." *Biometrika*, **44**:114–130, 1957.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications.* Springer, 1998.

[Mur12] Kevin Patrick Murphy. *Machine Learning: A Probabilistic Perspective.* MIT Press, 2012.

[NKR18] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying Properties of Binarized Deep Neural Networks." In *AAAI*, 2018.

[NYM17] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. "Compiling Graph Substructures into Sentential Decision Diagrams." In *AAAI*, pp. 1213–1221, 2017.

[OCD16] Umut Oztok, Arthur Choi, and Adnan Darwiche. "Solving PP^{PP}-Complete Problems Using Knowledge Compilation." In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 94–103, 2016.

[OD14] Umut Oztok and Adnan Darwiche. "On Compiling CNF into Decision-DNNF." In *CP*, pp. 42–57, 2014.

[PD11] Hoifung Poon and Pedro M. Domingos. "Sum-Product Networks: A New Deep Architecture." In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 337–346, 2011.

[Pea89] Judea Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference.* Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[PNK15] Johan Pensar, Henrik Nyman, Timo Koski, and Jukka Corander. "Labeled directed acyclic graphs: a generalization of context-specific independence in directed graphical models." *Data mining and knowledge discovery*, **29**(2):503–533, 2015.

[PSG09] Michal Piorkowski, Natasa Sarafijanovoc-Djukic, and Matthias Grossglauser. "A Parsimonious Model of Mobile Partitioned Networks with Clustering." In *The First International Conference on COMmunication Systems and NETworkS (COMSNETS)*, January 2009.

[PZ03] David Poole and Nevin Lianwen Zhang. "Exploiting contextual independence in probabilistic inference." *Journal of Artificial Intelligence Research*, **18**:263–313, 2003.

[RKG14] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. "Cutset Networks: A Simple, Tractable, and Scalable Approach for Improving the Accuracy of Chow-Liu Trees." In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, pp. 630–645, 2014.

[Rot96] Dan Roth. "On the Hardness of Approximate Reasoning." *Artif. Intell.*, **82**(1-2):273–302, 1996.

[SBZ06] Reid Simmons, Brett Browning, Yilu Zhang, and Varsha Sadekar. "Learning to predict driver route and destination intent." In *Intelligent Transportation Systems Conference*, pp. 127–132, 2006.

[SCD16] Yujia Shen, Arthur Choi, and Adnan Darwiche. "Tractable Operations for Arithmetic Circuits of Probabilistic Models." In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 3936–3944, 2016.

[SCD17] Yujia Shen, Arthur Choi, and Adnan Darwiche. "A Tractable Probabilistic Model for Subset Selection." In Gal Elidan, Kristian Kersting, and Alexander T. Ihler, editors, *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. AUAI Press, 2017.

[SCD18] Yujia Shen, Arthur Choi, and Adnan Darwiche. "Conditional PSDDs: Modeling and Learning With Modular Knowledge." In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pp. 6433–6442. AAAI Press, 2018.

[SCD20] Yujia Shen, Arthur Choi, and Adnan Darwiche. "A New Perspective on Learning Context-Specific Independence." In *10th International Conference on Probabilistic Graphical Models*, 2020.

[SGD19] Yujia Shen, Anchal Goyanka, Adnan Darwiche, and Arthur Choi. "Structured Bayesian Networks: From Inference to Learning with Routes." In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 7957–7965. AAAI Press, 2019.

[Shi94] Solomon Eyal Shimony. "Finding MAPs for Belief Networks is NP-Hard." *Artif. Intell.*, **68**(2):399–410, 1994.

[SM05] Scott Sanner and David A. McAllester. "Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference." In *IJCAI*, pp. 1384–1390, 2005.

[SM12] Charles Sutton and Andrew McCallum. "An Introduction to Conditional Random Fields." *Found. Trends Mach. Learn.*, **4**(4):267–373, April 2012.

[SMV19] Xiaoting Shao, Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Thomas Liebig, and Kristian Kersting. "Conditional Sum-Product Networks: Imposing Structure on Deep Probabilistic Architectures." *arXiv preprint arXiv:1905.08550*, 2019.

[STA12] Kevin Swersky, Daniel Tarlow, Ryan P. Adams, Richard S. Zemel, and Brendan J. Frey. "Probabilistic n-Choose-k Models for Classification and Ranking." In *Advances in Neural Information Processing Systems 25 (NIPS)*, pp. 3059–3067, 2012.

[SW93] Detlef Sieling and Ingo Wegener. "NC-Algorithms for Operations on Binary Decision Diagrams." *Parallel Processing Letters*, **3**:3–12, 1993.

[TJH05] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. “Large Margin Methods for Structured and Interdependent Output Variables.” *J. Mach. Learn. Res.*, **6**:1453–1484, December 2005.

[Vom06] Jirí Vomlel. “Noisy-or classifier.” *Int. J. Intell. Syst.*, **21**(3):381–398, 2006.

[Weg00] Ingo Wegener. *Branching Programs and Binary Decision Diagrams.* SIAM, 2000.

[XCD12] Yexiang Xue, Arthur Choi, and Adnan Darwiche. “Basing Decisions on Sentences in Decision Diagrams.” In *AAAI*, pp. 842–849, 2012.

[XZF18] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. “A Semantic Loss Function for Deep Learning with Symbolic Knowledge.” In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5502–5511, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[YFW05] Jonathan Yedidia, William Freeman, and Yair Weiss. “Constructing free-energy approximations and generalized belief propagation algorithms.” *IEEE Transactions on Information Theory*, **51**(7):2282–2312, 2005.

[Zhe15] Yu Zheng. “Trajectory Data Mining: An Overview.” *ACM Transaction on Intelligent Systems and Technology*, 2015.