HIGH PERFORMANCE ALGORITHMS FOR EXACT STRUCTURE LEARNING OF BAYESIAN NETWORKS

by

Subhadeep Karan
May 2019

A dissertation submitted to the Faculty of the Graduate School of the University at Buffalo, The State University of New York in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering
Copyright by
Subhadeep Karan
2019
The thesis of Subhadeep Karan was reviewed by the following:

Jaroslaw Zola  
Assistant Professor,  
Department of Computer Science & Engineering  
Department of Biomedical Informatics  
Thesis Advisor, Chair of Committee

Venu Govindaraju  
SUNY Distinguished Professor,  
Department of Computer Science and Engineering  
Committee Member

Varun Chandola  
Assistant Professor,  
Department of Computer Science and Engineering  
Committee Member
Dedication

To my wonderful parents Basanta Kumar Karan and Archana Karan, my brother Sandeep Karan and sister-in-law Soma Karan, who have been selflessly helping me pursue my dreams.
Acknowledgments

There are many people that have earned my gratitude for their contribution to my time in UB and without whom this thesis would not have been possible.

First, I am indebted to my advisor, Dr. Jaroslaw Zola. Since my first day in SCoRe group, Dr. Zola believed in me like nobody else and gave me endless support. He continually and convincingly conveyed a spirit of adventure in regard to research. It was him who took a leap of faith in me. He is an example of excellence as a researcher, mentor, instructor, and role model.

Second, I would like to thank my committee members, Dr. Venu Govindaraju and Dr. Varun Chandola, whose continuous support and engagement in my PhD work helped me to make constant progress. Also, I would also like to thank few Non-thesis Committee Members, Dr. Atri Rudra, Dr. Ajaya Mohanty, Dr. Abani Patra and Dr. Mohan Shetye who were always there to guide me when I needed any advise. I would also like to thank my lab mates for their continued support. This dissertation would not have been possible without the intellectual contribution of Frank Schoeneman, Vicky Zheng and Sai Vikneshwar.

I am also grateful to my industrial collaborators. I spent two summers at SAS
Institute where I had the chance to collaborate with fantastic researchers. More specifically, I would like to thank Ilknur Kaynar Kabul, Jorge Silva and Xin Hunt for providing me the great opportunity to work on innovative ML projects. I also extend my gratitude to members of the AI-ML and Cognitive Computing team at SAS for the fruitful discussions and for making my internship at SAS Institute such an amazing experience.

I would also extend my gratitude to the Center for Computational Research team at the University at for their unfailing technical and Hardware support.


This dissertation would not have been possible without their warm love, continued patience, and endless support.
# Table of Contents

**Acknowledgments**

**Abstract**

**Chapter 1**

  **Introduction**

**Chapter 2**

  **Preliminaries**

  2.1  **Notation and Terminology**

  2.2  **Learning Structure**

     2.2.1  **Efficient Data Access**

     2.2.2  **Parent Set Assignment Problem**

     2.2.3  **Structure Search**

  2.3  **Parameter Learning**

  2.4  **Equivalence Classes**
Chapter 3

Why Exact Structure Learning? 11

3.1 Introduction ................................................. 11

3.2 Testing Protocol ............................................ 14
  3.2.1 Data Fidelity .......................................... 15
  3.2.2 Learners Comparison Criteria ...................... 16

3.3 Experimental Analysis ................................. 17
  3.3.1 Fidelity ................................................. 19
  3.3.2 Inference Queries ..................................... 19
  3.3.3 Structure recovery ................................. 20

3.4 Conclusions ................................................ 21

Chapter 4

Efficient Data Access 23

4.1 Introduction ................................................. 23

4.2 Standard Approach ................................. 25

4.3 Proposed Approach .................................... 27
  4.3.1 Programming Interface ......................... 28
  4.3.2 Bitmap Strategy ..................................... 30
  4.3.3 Radix Strategy ....................................... 33

4.4 Experimental validation ............................... 35
  4.4.1 Queries in Bayesian Network Structure Learning (BNSL) . 36

4.5 Related work ................................................ 39

4.6 Conclusions ................................................ 40
Chapter 5

Parent Set Assignment Problem 42

5.1 Introduction .............................................. 43
5.2 Bayesian Network Structure Learning (BNSL) ............... 45
5.3 Proposed Approach ...................................... 46
  5.3.1 Constraining the Search Space ......................... 48
  5.3.2 Parallel Exploration .................................. 50
    5.3.2.1 Folding Lattices ................................. 51
    5.3.2.2 Limiting Synchronization ....................... 53
    5.3.2.3 Changing Exploration Mode .................... 55
  5.3.3 Apache Spark Implementation .......................... 56
5.4 Experimental Results .................................... 59
  5.4.1 Scalability Tests ................................... 60
  5.4.2 Hepar II Test ....................................... 63
5.5 Related Work ............................................ 64
5.6 Conclusions ............................................. 65

Chapter 6

Optimal Path Extension 67

6.1 Introduction ............................................. 67
6.2 Proposed Method ....................................... 69
  6.2.1 Optimization Problem ............................... 70
  6.2.2 Optimal Path Extension ............................ 72
  6.2.3 Searching via Optimal Path Extension ............... 76
6.3 Experimental Results .................................... 79
6.3.1 Test Data and Experimental Setup ....................... 80
6.3.2 Results Discussion ........................................... 82
6.4 Conclusions ..................................................... 84

Chapter 7

Conclusions and Open Problems 85
7.1 Conclusions ..................................................... 85
7.2 Open Problems .................................................. 86

Bibliography 88
Abstract

Bayesian networks (BNs) are probabilistic graphical models often used in big data analytics to capture conditional relationships among a set of random variables. They are critical class of models that are capable to represent both association and causation, and hence are indispensable in building AI systems. Over the years, BNs have been successfully applied in many domains including diagnostic systems, clinical decision support, systems biology, and genomics. However, the problem of learning structure of BNs from data, which is a typical starting point in BNs applications, is known to be NP-hard. To date, both heuristics and exact learning algorithms have been proposed to tackle the problem. While heuristics are widespread in real-life applications, exact algorithms are believed to learn better networks at the much higher computational cost.

In this work, we first show that compared to heuristics the exact structure learning algorithms are delivering measurably better structures in terms of Structural Hamming Distance, and enable more accurate inferences, measured through the Kullback-Leiber divergence between the actual and inferred probability distributions. Then, we propose high performance strategies to address computational challenges of exact structure learning algorithms.
The problem of learning exact BN structure is a data-driven optimization problem that involves three key subproblems: (i) counting queries, to evaluate objective function, (ii) parent sets identification to establish local dependencies between variables, and (iii) structure search, to combine locally optimal solutions into globally optimal structure. In this work, we propose efficient approach for each subproblem. Specifically, we design programming abstraction and algorithmic realizations for fast counting in machine learning applications, which offers order or magnitude higher throughput of counting queries than the existing techniques. We then introduce a new shared and distributed memory approach for the exact parent sets assignment problem. To achieve scalability, we derive theoretical bounds to constrain the search space when MDL scoring function is used, and we reorganize the underlying dynamic programming problem such that the computational density is increased and fine-grain synchronization is eliminated. We demonstrate that the resulting method maintains strong scalability on large Apache Spark clusters, and it can be used to efficiently process data sets with over 70 variables, far beyond the reach of the currently available solutions. To address the structure search problem, we reduce it to a single source shortest path problem in partial order lattices. Because the resulting graphs grow exponentially with the number of input variables, we introduce a new approach that exploits partial relationships between variables to constrain the number of ways in paths in the graph can be extended, while remaining provably optimal. Via experimental results, we demonstrate that the method provides up to three times improvement in runtime, and orders of magnitude reduction in memory consumption over the current best algorithms. The
proposed algorithms are implemented in the open source SABNA package – a high-quality software package that can be used by researchers and practitioners to analyze complex data sets, leveraging parallelism of modern shared-memory multi-core processors.
Introduction

Bayesian networks (BNs) are a class of probabilistic graphical models that capture conditional relationships among a set of random variables. In BNs, the relationship between variables is qualitatively described by conditional independencies, and quantitatively assessed by conditional probability distributions. BNs serve as a powerful tool for structuring probabilistic information and hence are an ideal framework for complex inferences including predictive, diagnostic and explanatory reasoning [1].

In many real-world applications, BNs outperform more sophisticated machine learning methods (including deep learning) [2, 3, 4, 5, 6, 7], or are desired because they are generative, support speculative queries, and are relatively easy to interpret. For instance, in system biology, BNs are built from gene expression data and are directly used to analyze potential regulatory interactions between genes. This is because BN structure explicitly and with high accuracy represent gene interactions. In clinical decision support systems, BNs are frequently constructed from the Electronic Health Records (EHRs) to obtain the most likely di-
agnosis and prognosis of patients. This is possible as BNs support explanatory and speculative queries and uniquely incorporate prior knowledge. In banking, BNs are used for fraud detection and are favored for the transparency and clarity in representing the reasoning chain that led to flagging given transaction as fraudulent.

BNs are especially important tool in biomedical informatics. Searching for term Bayesian Network in just PubMed (the primary aggregator of publications in life sciences), reveals over 2,700 publications in the last five years alone. In the overwhelming majority of these publications, probabilistic Bayesian modeling has directly influenced research findings. This is not surprising, considering the growing importance of data-driven reasoning in medicine, and life sciences in general [8]. As more data becomes available, the need to reliably organize it for reasoning will be growing, and hence high quality Bayesian models will be essential. The same argument holds true for much broader spectrum of application domains. For example, companies like AgenaRisk [9], Bayesia [10], Bayes Server [11] or HuginExpert [12], along with many others, created solid enterprises around Bayesian networks in areas of risk assessment, strategic planning and fraud detection. In all these cases, high quality Bayesian models have been the key to success.

Currently, the state of the art AI systems operate in a pure statistical, or model-free, mode. They rely only on symmetric associations between variables, are in the essence are able to capture (sometimes extremely complex) correlations, but are not designed to tackle causation. Yet, the ability to express causation is the key to perform in silico interventions, which in turn enable retrospections [13]. And, as argued by Judea Pearl [13] (father of Bayesian networks), retrospections are the pillars for building a system having human-level intel-
ligence. Such a system should be developed based on the three-layer casual hierarchy: (i) Association (ii) Intervention and (iii) Counterfactual.

In the association layer, the simpler asymmetric relationships are learned and expressed in the format of $P(y|x) = p$ stating that: the probability of event $Y = y$ given that we observed event $X = x$ is equal to $p$. Such relationships can be learned from the observed data. Intervention layer which is built upon association layer provides more richer expression in the form $P(y|do(x), z)$, which denotes the probability of event $Y = y$ given that we intervene and set the value of $X$ to $x$ and subsequently observe event $Z = z$. To learn such relationships, one has to deploy causal Bayesian networks [14] or randomized trials (which are typically impossible to perform). Counterfactuals [13] are the highest level in the causal hierarchy. In this layer, expressions have the characteristics of being imaginative and retrospective, involving queries like Why? and Was it $X$ that caused $Y$?. Thus to build a true AI system, we must be able to construct causal networks on top of the basic associations in the data. But a causal network is in the essence a high quality Bayesian network in which edges are trusted to represent truly causal relationships. By definition of causality, such network will always be encoding human understanding of the physical world. However, in complex systems, capturing such causal effects will usually require to first organize the probabilistic information about its components (i.e., variables).

While BNs are recognized as a powerful method for structuring probabilistic information and are the platform of choice in situations where complex inferences have to be performed [1], they face a significant challenge. This is because the critical components in BNs modeling, i.e., structure learning from data, and inference (see Figure 2.1), are both known to be $NP$-hard [15, 16, 17, 18, 19, 20]. Consequently, the current state of the art depends on heuristics [21, 22, 23, 24,
However, by their nature, heuristics do not provide any guarantees about the quality of the structures they find or precision of inferences they make. When network structure is learned from relatively small data, heuristics easily fall into local optima resulting in sub-optimal or incorrect models. When large or big data is available, the problem persists, since typically with more available data more variables are added to the model. This weakness of heuristics percolates to ensemble learning and model averaging, where multiple models are combined into a (potentially) better single model. Here the uncertainty of the component networks contributes to the uncertainty of the final model. As a result, the uncertainty due to the data cannot be separated from the uncertainty of the learning and inference algorithms. Finally and most importantly, heuristics often involve stochastic processes and are sensitive to the choice of model-unrelated parameters. Consequently, they impact reproducibility since different executions of the same heuristic may lead to significantly different outcomes. This is a critical issue taking into account the fundamental role of reproducibility in science and engineering. The above shortcomings are addressable if heuristics are replaced by exact, i.e., provably optimal, approaches.

In this thesis, we decompose the problem of exact Bayesian network structure learning (BNSL) into three core sub-problems: (i) efficient data access, (ii) parent set assignment, and (iii) structure search. To make the BNSL workflow scalable, we address each of the sub-problems separately by developing new algorithms and data representation techniques, tailored for modern parallel architecture.

The organization of the thesis is as follows: Chapter 2, provides the notations, terminology which we use across the thesis. Chapter 3, presents a princi-
pled and systematic approach to evaluate exact methods against heuristic methods. In Chapter 4, we provide two new efficient counting query approaches tailored for machine learning (ML) applications, including BNSL. In Chapter 5, we present a scalable approach for parents sets assignment, that has nearly linear speedup of distributed memory and shared memory systems. Chapter 6 presents the structure search procedure along with a new optimization allowing to realize exact solutions for some of the largest networks for which no exact solution exists.
Preliminaries

2.1 Notation and Terminology

Formally, BN over a set of \( n \) random variables \( \mathcal{X} = \{X_1, X_2, \ldots, X_n\} \) is a pair \( N = (G, \theta) \), where \( G \) is a directed acyclic graph (DAG) encoding conditional independencies between \( \mathcal{X} \) with respect to the probability \( \theta \) that factorizes over \( G \). Figure 2.1 is a depiction of different stages involved when working with BNs.

![Figure 2.1: Simplified representation of probabilistic modelling with BNs.](image)

We focus on the common case of BNs with categorical random variables where variable \( X_i \) take values \( x_i \) from the domain \( \text{Dom}(X_i) \) of size \( r_i \). Domain of variable \( X_i \) is represented by states \([x_{i1}, \ldots, x_{ir_i}]\). Alternatively, we can think of \( X_i \) as a symbolic feature with arity \( r_i \), and for convenience we can represent
its states by integers \([1, \ldots, r_i]\). Let \(D = [D_1, D_2, \ldots, D_n]\) be a complete database of instances of \(\mathcal{X}'\), where \(D_i, |D_i| = m\), records observed states of \(X_i\) as depicted in Figure 4.1.

Since variables \(X_i\) are categorical, we assume that \(\theta\) is a multinomial distribution, and represent it via a set of local conditional probability tables. Specifically, table \(\Theta_{X_i|Pa(X_i)}\) maintains conditional probabilities \(P(x_i | (x_j)_{X_j \in Pa(X_i)}) = \theta_{x_i|(x_j)_{X_j \in Pa(X_i)}}\) for all \(r_i\) possible configurations of \(X_i\), and \(q_i = \prod_{X_j \in Pa(X_i)} r_j\) configurations of its parents, \(Pa(X_i)\), in \(G\). We will use \(|\Theta|\) to denote the number of parameters in table \(\Theta\).

### 2.2 Learning Structure

For the given \(D\), we can learn a BN structure \(G\) directly from the data. Our focus is on score-based learning methods, in which structure learning amounts to optimizing a function that measures goodness of fit between \(G\) and \(D\).

Let \(Score(G:D)\) be a scoring function evaluating quality of the network structure \(G\) with respect to the input data \(D\). Furthermore, let \(Score(G:D)\) be decomposable, that is:

\[
Score(G:D) = \sum_{X_i \in \mathcal{X}} s(X_i, Pa(X_i)),
\]

where \(s(X_i, Pa(X_i))\) is a score contribution of \(X_i\) when its parents are \(Pa(X_i)\).

Examples of such scoring functions include popular BIC [43] and MDL [43] derived from information theory or BD [44] and BDe [45] that implement Bayesian scoring criteria. Decomposability is commonly assumed to improve the search process as local changes to a network structure can be evaluated quickly.
2.2.1 Efficient Data Access

To develop an efficient solver for the parent set assignment problem, it is necessary to compute the scoring functions efficiently which in turn depends on the efficiency of the counting queries. The counting query $\text{Count}((X_i = x_i) \land (X_j = x_j) \land \ldots)$ returns the size of the support in $D$ for the specific assignment $[x_i, x_j, \ldots]$ of variables $[X_i, X_j, \ldots]$ to do that it access dataset. For example, for the database in Figure 4.1, the query $\text{Count}((X_1 = 3) \land (X_2 = 2) \land (X_3 = 1))$ would return 2, as there are 2 instances matching the query condition. We note that the above formulation of counting is a special and simple case of the general counting problem in conjunctive queries, known from database theory [46] (we provide more details in Section 4.5).

In the context of BNSL, computation of the scoring function $s(X_i, Pa(X_i))$ invokes counting queries as explained in Section 4.2.

2.2.2 Parent Set Assignment Problem

The parent set assignment problem is enumerate all possible local structure and enumerate them. Which is reduced to problem finding the subset $Pa(X_i) \subseteq \mathcal{X} - \{X_i\}$ such that $s(X_i, Pa(X_i))$ is minimized.

Let $d(X_i, U), U \subseteq \mathcal{X} - \{X_i\}$, be the score of selecting optimal parent set of $X_i$ from among variables in $U$, that is $d(X_i, U) = \min_{Pa(X_i) \subseteq U} s(X_i, Pa(X_i))$. We can efficiently express $d$ via the following recursion:

$$d(X_i, U) = \min \begin{cases} s(X_i, U), \\ \min_{X_j \in U} d(X_i, U - \{X_j\}). \end{cases}$$

(2.1)
To find an optimal parent set assignment of $X_i$ we could solve the recursion in Equation 2.1 for $U = \mathcal{X} - \{X_i\}$ while recording the choice of parents we made in the process.

### 2.2.3 Structure Search

Any DAG with nodes $\mathcal{X}$ can be equivalently represented via one of its topological orderings of $\mathcal{X}$. A topological ordering implies that $X_i$ is always preceded by $X_j$, written as $X_j \prec X_i$, if $X_j$ is a parent of $X_i$, i.e. $X_j \in Pa(X_i)$. Let $\pi(U)$ denote a topological ordering over a set $U \subseteq \mathcal{X}$.

Then, we can define $Q(U, \pi)$ as:

$$Q(U, \pi) = \sum_{X_i \in U} d(X_i, \{X_j | X_j \prec X_i \text{ in } \pi(U)\})$$  \hspace{1cm} (2.2)

where $Q(U, \pi)$ is a score of an ordering (i.e. network) over $U$, $\{X_j | X_j \prec X_i \text{ in } \pi(U)\}$ is the set of the preceding variables in the given ordering.

Using parent set assignment solver as a subroutine described in Section 6.2, allows efficient memoization of $d$. It also allows several opportunities for developing new optimizations and also abstracts away the choice of scoring function from the structure search module.

### 2.3 Parameter Learning

For the given network structure $G$ and $D$, we obtain final BN model by estimating probability $\theta$ from the input data $D$. Parameters $\theta_{x_i(x_j)}$ can be directly obtained from $D$ via Maximum Likelihood Estimation (MLE) or Maximum Posterior Approximation (MAP) [47]. Both MLE and MAP are easy to
perform provided that we have sufficiently large $D$, and efficient mechanism to run counting queries over $D$.

2.4 Equivalence Classes

Two network structures are equivalent if they represent the same set of probability distributions. To represent equivalent networks, we can use a partially directed acyclic graph (PDAG), in which directed edges correspond to the edges that have the same orientation in every network in the same equivalence class, and undirected edges correspond to edges that may have different orientation in different networks in the same class [48]. Consequently, if $\text{pdag}(G_i) = \text{pdag}(G_j)$, where $\text{pdag}(G)$ denotes PDAG of structure $G$, then $G_i$ and $G_j$ belong to the same equivalence class and can be considered indistinguishable.
Why Exact Structure Learning?

Overview. Learning structure of BN from data is a common task in Machine Learning applications. The problem is known to be NP-hard, and both heuristics and exact learning algorithms have been proposed. While heuristics are widespread in real-life applications, exact algorithms are believed to learn better networks. In this chapter, we ask and answer the question of whether exact learning algorithms offer practical advantage over popular heuristic used for BNSL.

3.1 Introduction

In many practical applications, for example in computational biology and biomedical informatics [49], the structure of BN, is not known a priori and has to be learned from data. This task can be performed either by optimizing a scoring function that expresses quality of a structure given the data (score-based learning), or by assessing conditional independence constraints directly from the data (constraint-based learning) [50]. However, irrespective of the selected learning
strategy, the problem is known to be NP-hard, even under simplifying assumptions, like bounded in-degree of nodes in the structure [15, 51] (although we should note that variants that are not NP-hard do exist [52, 53]). Yet, in spite of this discouraging result, over the years multiple exact structure learning algorithms have been proposed (see, e.g., [54, 55, 56, 57, 58, 59] to get a sample of the most recent developments).

Exact algorithms are computationally and memory expensive, and inherently limited to problems with at most several dozen variables (even when scaled using parallel computing [60, 61, 62]). Consequently, fast and easily accessible heuristics, like Tabu Search (TS) and Hill Climbing (HC), have been favored by practitioners in real-life applications [63]. Naturally, heuristics do not provide any guarantees about structures they find. In fact, the common argument motivating development of the exact structure learning algorithms is that optimal network structure leads to the better quality final model, and reduces model uncertainty, as compared to the structures learned using heuristics. But, to the best of our knowledge, this argument has been primarily based on anecdotal evidence. This leads us to the main question of the chapter: Do exact learning algorithms offer practical advantage over popular heuristic used for BNs structure learning?

In the recent study, Scutari et al. ask somewhat related question: “Who learns better BN structures?” [50], and compare score-based and constraint-based methods. The study does an excellent job of rigorously separating structure search from structure evaluation across different learners. It also factors in the computational cost of structure learning, and considers a broad spectrum of test cases. However, the comparison does not include exact algorithms, and focuses solely on the qualitative description of recovered structures, ex-
pressed as the fraction of correctly identified edges with respect to the reference ground truth model. In that sense, the study is not conclusive: without exact algorithms, differences between score-based and constraint-based methods (reported by Scutari et al.) cannot be attributed solely to structure scoring criteria, as they could result from the use of sub-optimal search strategies. Furthermore, without using learned BN in inference, it is difficult to decide whether the structure is actually good (see, e.g., [64]).

In this chapter, we approach the question of Where are exact methods preferred over heuristics?. We explicitly focus on score-based exact algorithms and heuristics, and we look at structure learning both qualitatively, by assessing the capability of different learners to recover the underlying structure, and quantitatively, by checking how recovered structures affect the inference of posterior probabilities. The resulting benchmarking strategy aims to mimic the typical applications of BNs which typically involve reasoning through inferences. We then propose an intuitive experimental protocol based on exact inference method, which we follow to conduct an extensive empirical evaluation on multiple popular benchmark data. The use of exact inference [65] method makes it possible to compare different learners reliably, and it ensures the reproducibility of the protocol.

This chapter is organized following the common convention. In Section 3.2 we discuss the proposed testing criteria in detail. We describe the experiments and results in Section 3.3, and conclude the chapter with brief discussion in Section 3.4.
3.2 Testing Protocol

To quantify the performance of different learning methods, we designed the protocol outlined in Figure 3.1. We start with some benchmark BN, denoted by $N^* = (G^*, \Theta^*)$, which we use as a ground truth model to generate data, $D$, with respect to which we will be assessing learners. By sampling data from such reference distribution we are ensuring that the data can indeed be described by a BN, and that we know the actual structure, $G^*$, of that network. When sampling, we will consider different ratios $\rho = \frac{m}{|\Theta^*|}$, where $m$ is the number of observations in $D$. This is to account for different use scenarios, where the input data can be considered scarce ($\rho < 1$) or abundant ($\rho > 1$). Here we anticipate that the data availability will directly affect learners performance.

Once the data $D$ is generated, we will use selected learners to obtain the test networks $N_i = (G_i, \Theta_i)$. For the comparisons, we selected three particular methods: two heuristics, Tabu Search and Hill Climbing, which both are classic optimization methods with efficient implementation in the popular bnlearn package [66], and one exact method, SABNA, which is high performance open source exact BN learning tool [58]. We opted for these particular methods by
assuming the end-user point of view: the considered methods are easily accessible and have well documented software realizations. At the same time we recognize that other heuristics, e.g., [67], would be interesting to include in the comparison, but their corresponding software is currently not available.

3.2.1 Data Fidelity

For the given network $N^* = (G^*, \theta^*)$, $\rho$ and the sample $D$ quantify how faithfully the given sample captures the underlying signal (probability distribution) of the actual network. Quantifying the fidelity of the data helps in getting more insight in identifying the scenarios where specific structure learning algorithms can be expected to perform well and where not. For samples with lower fidelity score, the structure learning is expected to be difficult which should be reflected via learners comparison criteria defined in Section 3.2.2. It is known that the sample size to faithfully represent the distribution increases as the underlying distribution becomes more sparse.

Fidelity score of the data also allows verifying the hypothesis whether with larger sample size the data becomes more faithful or not. Also, whether the exact structure learning method becomes invariant to increment in the sample size or not, results and analysis for this are delegated to our repository [https://gitlab.com/SCoRe-Group/bn-sl-compare](https://gitlab.com/SCoRe-Group/bn-sl-compare).

To compute the fidelity score of $D$, we use the ground truth network structure $G^*$ and learn the parameters $\Theta'$ from $D$. $N' = (G^*, \theta')$ is used as the reference network. Under the ideal condition, the network $N_i$ learned from the given sample $D$ is equivalent to $N'$. We define the fidelity of the data as:
\[ fidelty(D, \theta^*, \theta') = \frac{\sum_{X_i} KL_D \left( \Theta^*_{X_i|Pa(X_i)} \| \Theta'_{X_i|Pa(X_i)} \right)}{n} \] (3.1)

### 3.2.2 Learners Comparison Criteria

When comparing our selected learners we will consider two main criteria. The first is how close is the learned structure \( G_i \) to the reference structure \( G^* \). This is very common comparison criterion, and it is typically based on so called Structural Hamming Distance (SHD) [37]. In essence, SHD compares how many edges are different between \( \text{pdag}(G^*) \) and \( \text{pdag}(G_i) \). The use of PDAGs instead of the actual structures is essential to eliminate statistically insignificant differences between the structures. In our protocol, we take slightly more fine grained view of network structure, and differentiate between edges that are incorrectly present and are incorrectly missing in \( G_i \) compared to \( G^* \). Let \( E(G) \) be a set of edges of graph \( G \). We will consider precision, defined as:

\[
\text{precision}(N^*, N_i) = \frac{|E(\text{pdag}(G^*)) \cap E(\text{pdag}(G_i))|}{|E(\text{pdag}(G_i))|},
\]

and recall, defined as:

\[
\text{recall}(N^*, N_i) = \frac{|E(\text{pdag}(G^*)) \cap E(\text{pdag}(G_i))|}{|E(\text{pdag}(G^*))|},
\]

and we will summarize the two into a single statistics using \( F_1 \) score, defined as:

\[
F_1(N^*, N_i) = \frac{2 \cdot \text{precision}(N^*, N_i) \cdot \text{recall}(N^*, N_i)}{\text{precision}(N^*, N_i) + \text{recall}(N^*, N_i)}.
\]

For the second criteria, we will be looking into the accuracy of inferences
performed on the resulting models. Let $\Theta^N_Q$ denote the conditional probability table obtained by executing inference query $Q$ over the network $N$. The query $Q$ will be always clear from the context, and it will be characterized by the variables to be inferred. We compare two networks, $N_i$ and $N_j$, by computing the Kullback-Leibler (KL) divergence from $\Theta^N_{Qi}$ to $\Theta^N_{Qj}$, defined as:

\[
D_{KL}(\Theta^N_{Qi}||\Theta^N_{Qj}) = \frac{\sum_x \theta^N_{xi} \log \left( \frac{\theta^N_{xi}}{\theta^N_{xj}} \right)}{n}.
\]

If $N_j$ is a reference network (e.g., a known ground truth model), we can interpret KL divergence as the information gain we could achieve if the inference was made over the reference model instead of $N_i$. Hence, we can intuitively expect that a good learning method will deliver networks with KL divergence close to zero when compared to the ground truth, across all queries of interest.

### 3.3 Experimental Analysis

We ran all our experiments on a server with two Intel Xeon E5 – 2650 2.30GHz 10-core CPUs, and 64 GB of RAM. To perform experimental analysis we use the set of standard benchmark BNs available from [66]. In what follows, we will discuss only networks summarized in Table 3.1, and we refer reader to [https://gitlab.com/SCoRe-Group/bn-sl-compare](https://gitlab.com/SCoRe-Group/bn-sl-compare) where we provide results for other networks, together with a complete set of software tools to reproduce our tests.

We specifically assume the MDL scoring function [43] defined as $\text{Score}(G:D) = \sum_{i=1}^{n} \left( \log(P(X_i|Pa(X_i)) - \frac{[\Theta_{X_i}]}{2}\log(m) \right)$. This choice is motivated by practical considerations: MDL is arguably the most frequently used scoring function in real-world applications (see, for example, [68]), and it does not involve hyper-
parameters (unlike, e.g., BD or BDeu functions [69]). This makes it more amenable to experimental analysis. Moreover, MDL has been demonstrated to correlate with $G^2$ independence test, which is frequently used in constraint-based learning [50, 70] (hence, we can hope to generalize some of the findings to constraint-based methods, similar to [50]). Finally, to parameterize networks, we will be using the standard Maximum Likelihood Estimation approach (MLE), as we assume no prior knowledge about the distribution $P$.

We use three real networks from the BN Repository [66]. We consider a sufficiently large set of $\rho$ values and for each fixed $\rho$, we consider 100 samples of $D$. Here, we present only the representative $\rho$ values here. We describe the parameters of these networks in detail in Table 3.2. Further, in each sample for every $(N^*, \rho)$, we ensure that each state for every variable in $G^*$ is present. In other words, we reject samples that do not meet this criterion. We would like to mention here that is this is an accepted procedure [71]. We have hosted all of our results along with additional datasets to our repository https://gitlab.com/SCoRe-Group/bn-sl-compare
3.3.1 Fidelity

Figure 3.2 summarizes the experiment as it shows that for lower $\rho$ the data is less faithful and improves with increase in the sample size. For “Insurance,” the fidelity score remained very low for $\rho < 150$. On evaluation, it was found that the probability distribution associated with it is much sparser as compared to other datasets. As a result, it requires relatively larger sample.

```
<table>
<thead>
<tr>
<th>$\rho$</th>
<th>Child</th>
<th>Insurance</th>
<th>Alarm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td><img src="image1" alt="Box plot" /></td>
<td><img src="image2" alt="Box plot" /></td>
<td><img src="image3" alt="Box plot" /></td>
</tr>
<tr>
<td>0.75</td>
<td><img src="image4" alt="Box plot" /></td>
<td><img src="image5" alt="Box plot" /></td>
<td><img src="image6" alt="Box plot" /></td>
</tr>
<tr>
<td>1.5</td>
<td><img src="image7" alt="Box plot" /></td>
<td><img src="image8" alt="Box plot" /></td>
<td><img src="image9" alt="Box plot" /></td>
</tr>
<tr>
<td>3.0</td>
<td><img src="image10" alt="Box plot" /></td>
<td><img src="image11" alt="Box plot" /></td>
<td><img src="image12" alt="Box plot" /></td>
</tr>
<tr>
<td>6.0</td>
<td><img src="image13" alt="Box plot" /></td>
<td><img src="image14" alt="Box plot" /></td>
<td><img src="image15" alt="Box plot" /></td>
</tr>
</tbody>
</table>
```

Figure 3.2: Box plot representing the distribution of the fidelity score for different $\rho$.

3.3.2 Inference Queries

We restrict ourselves to variable marginals queries only. This decision comes from the fact that they are relatively cheaper to compute. They are also well-
studied queries from the perspective of computational complexity. The reference network $N'$ may differ from the learned structure $N_i$, and as a result $\theta'$ and $\theta_i$ may not be comparable via Equation 3.1. Intuitively, its expected that performance of the inference queries for the data which with higher fidelity score should be better than the others.

Figure 3.3 summarizes this experiment. As expected, $D_{KL}(\Theta_{Q}^{N'} || \Theta_{Q}^{N_i})$ associated with exact method are closer to zero as compared to heuristics. The performance difference is very much prominent for lower $\rho$ values where the structure learning is expected to be difficult, suggesting that heuristics are far less trustable when the sample size is small. Difference in performance is compounded if the underlying distribution is less uniform and more sparse as depicted in the case of ”Insurance.”

3.3.3 Structure recovery

Our next of experiments concern structure recovery. Results of these experiments are presented in Figure 3.4. It’s quite evident that exact method outperforms the other heuristic methods in recovering the actual edge relationships. Unlike in the results reported in Section 3.3.2, even in the presence of more data the heuristic fails to bridge the difference compared with the exact. From these results, we can confidently conclude that if the primary aim of using structure learning algorithm is to learn edge relationship among variables, then BN obtained from heuristics are likely to be inferior than exact methods.
Figure 3.3: Comparison of SABNA, Tabu Search and Hill Climbing on sampled datasets. Scatter plots show how the output of different structure learning methods affect the inference. Y-axis: $D_{KL}(\Theta_N^Q || \Theta_{N_i}^Q)$ where $Q$ represents the variable marginal queries. X-axis: Fidelity score of the corresponding sample.

- SABNA;
- TABU;
- Hill Climbing

3.4 Conclusions

Our results show that exact learning methods are more robust to the sample size, both in terms of recovering the underlying structure as well as the distribution. They also indicate that the quality of posterior probability are less susceptible to the quality of the structure when sufficient data is available. Thus, our results have immediate practical value: in cases when the motive is interpreting the interaction among the variables via learned edge relationship or the sample size is small, heuristics should be avoided and slower but more robust
Figure 3.4: Comparison of SABNA, Tabu Search and Hill Climbing on sampled datasets. Scatter plots shows the performance of different structure learning methods in recovering the underlying structure. Y-axis: $1 - F_1(N^*, N_i)$. X-axis: Fidelity score of the corresponding sample.

- SABNA; TABU; Hill Climbing

exact methods should be preferred.
Overview. We propose scalable methods to execute counting queries in machine learning applications. To achieve memory and computational efficiency, we abstract counting queries and their context such that the counts can be aggregated as a stream. We demonstrate performance and scalability of the resulting approach on random queries, and through extensive experimentation using BNSL. Our methods significantly outperform commonly used ADtrees and hash tables, and are practical alternatives for processing large-scale data.

4.1 Introduction

Counting data records with instances that support some specific configuration of the selected variables is one of the basic operations utilized by machine learning (ML) algorithms. However, here we will focus in its usage for BNSL, data counting is necessary to evaluate a scoring function, or to assess constraints (e.g., via mutual information) [1].
While counting is typically viewed as a black-box procedure, and implemented using simple and not necessarily efficient strategies, e.g., contingency tables, in many practical applications it accounts for over 90% of the total execution time (we show a practical case in Section 4.4). Consequently, improving performance of counting can directly translate into better both in terms of scalability and runtime performance of BNSL. At the same time, popular specialized approaches based on data indexes, such as ADtrees [72], have limited applicability due to the significant preprocessing and memory overheads, which easily exceed the capability of current computational servers. This holds true for a broad spectrum of problem sizes and applications, with cases involving anywhere from tens to hundreds of variables, and thousands to millions of instances. As the size of the data analyzed by ML codes increases, there is a clear need for easy-to-adopt, efficient and scalable counting strategies.

In this dissertation, we address the above challenge by designing simple, yet fast and memory efficient counting strategies. Our methods are derived from the standard techniques like bitmap set representation and radix sorting, which can be efficiently implemented in a software. We describe an intuitive and convenient programming interface that leverages properties of the operators used in ML to separate the counting process from how the counts are utilized. This interface enables us to aggregate counts in a stream-like fashion. We encapsulate our methods in an open source software, and demonstrate its performance BNSL. Through extensive experiments on multiple popular benchmark datasets, we show that our strategies are orders of magnitude faster than the commonly used methods, such as ADtrees and hash tables.

This chapter is organized following the common convention. In Section 4.2 we discuss the standard approaches used till date for answering. We propose
different strategies for counting queries and discuss their pros and cons in Section 4.3. Experimental setup along with the results are discussed in Section 4.4. We close the paper with a brief survey of related work in Section 4.5, and concluding remarks in Section 4.6.

### 4.2 Standard Approach

In parent set assignment problem which is one of the core routines in BNSL, counting queries may account for over 90% of the total execution time and a group of consecutive queries is executed over the same set of variables (i.e., the queries share context). For instance, consider log-likelihood score frequently used in BNSL [73]:

\[
\mathcal{L}(X_i|Pa(X_i)) = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \left( \frac{N_{ijk}}{N_{ij}} \right),
\]

where \( Pa(X_i) \subseteq \mathcal{X} - \{X_i\} \) is a set of predictor variables for \( X_i \), \( j \) enumerates all possible \( q_i = \prod_{X_j \in Pa(X_i)} r_j \) states of variables in \( Pa(X_i) \), and \( N_{ij} \) and \( N_{ijk} \) are respectively the counts of instances in \( D \) such that variables in \( Pa(X_i) \) are in state \( j \), and the counts of instances such that variables in \( Pa(X_i) \) are in state \( j \) and \( X_i \) is in state \( k \). To compute \( \mathcal{L} \) we require multiple counting queries over the same group of variables \( Pa(X_i) \cup \{X_i\} \), testing different configurations of their states. Moreover, we care only about queries that return non-zero counts \( N_{ijk} \) (note that non-zero \( N_{ijk} \) implies non-zero \( N_{ij} \)), since only those contribute to the final sum.

The standard approach to handle queries that share context is to either directly scan the database \( D \) to construct \( r_i \times q_i \) contingency table of counts (or its
high-dimensional variant such as data cube [74]), or to first create an ADtree index to cache all sufficient statistics from $D$, and then to materialize contingency table on demand. Here materialization is done by retrieving the required counts via fast traversal over the index [72, 75]. However, both these approaches have significant limitations.

To use a contingency table we have to either maintain a lookup table with $r_i \cdot q_i$ entries, or to use a dictionary (e.g., hash table) with keys over the states of $Pa(X_i)$ and values being vectors of counts for the corresponding states of $X_i$. While lookup table may offer very fast memory accesses during construction and querying phases, it becomes computationally impractical, since usually it is very sparse. This is because even for large $m$, most of the time $D$ will not contain all $q_i$ possible configurations for the majority of sets $Pa(X_i)$. Consequently, lookup tables become a feasible choice only when we are dealing with a small number of variables, each with very small arity. Dictionaries address the problem of sparsity, as they store only configurations that are observed in $D$. However, they impose non-trivial overheads owing to the cost of hashing in a hash table dictionary or traversing scattered memory in a search tree dictionary. Moreover, when large number of high-arity variables are considered, a dictionary quickly becomes memory intensive easily exceeding capacity of a typical cache memory.

The alternative approach is to use one of many published variants of the ADtree index, e.g., [72, 75, 76, 77]. Here the idea is to first invest (significant) time and memory to enumerate and cache counts of all configurations found in $D$, and then reference those counts to answer subsequent queries. However, even with various optimizations, the space complexity of ADtrees is exponential in the number of variables, and even for modestly sized $D$ it may exceed
the available main memory. Moreover, by caching all counts indiscriminately, ADtrees often store entries that are never referenced in a given application, creating unnecessary memoization and searching overhead. Finally, ADtrees still require that a contingency table is materialized to deliver retrieved counts, and hence they pose a significant challenge in balancing memory and computations.

4.3 Proposed Approach

Given the database $D$, our goal is to provide memory and computationally efficient mechanism to answer counting queries with shared context. The memory efficiency is critical, since parent set assignment problem already have significant memory constraints (see for example [54]). If the memory has to be devoted to handling queries instead of being used by the actual algorithm, it would clearly constrain the applicability of the algorithm. At this point it is worth noting that similar ML applications fall into a gray zone in terms of the size of the input data on which they typically operate. On the one hand, the size of the input is too small to benefit from many excellent optimizations known from database theory (some we review in Section 4.5), as those are targeting cases in which volume of the data necessitates concurrent use of both persistent and main memory. On the other hand, the data is too large to warrant efficient execution using direct techniques like simple contingency tables.

To address this situation, we first define an intuitive programming interface to abstract the query context, including how counts are utilized by the target application. Then, we overlay the interface on top of two simple, yet very efficient, query execution strategies, where instead of storing counts we *consume* them in a stream-like fashion.
<table>
<thead>
<tr>
<th></th>
<th>D₁</th>
<th>D₂</th>
<th>D₃</th>
<th>X₁</th>
<th>X₂</th>
<th>X₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[X_1 = 1 \land X_3 = 1\]

**Figure 4.1:** (a) Database \( D \) with three variables, and the corresponding bitmap representation of \( X_1 \). (b) Example of executing Query \((X_2, \{X_1, X_3\})\) over \( D \) using Bitmap strategy, and (c) Radix strategy.

```cpp
class L {
public:
    void operator() (int Nijk, int Nij) {
        double p = 1.0 * Nijk / Nij;
        score += (Nijk * log2(p));
    }
    double score() const { return score_; }
private:
    double score_ = 0.0;
};
```

**Figure 4.2:** Example C++ code packaging Equation 4.1 into our programming interface.

### 4.3.1 Programming Interface

In a typical application, counts provided by queries with shared context are iteratively aggregated via some associative and commutative operator. One simple example with the summation operator is given in Equation 4.1. A more complex example could be Dirichlet priors with the product of gamma functions [44]. From the computational point of view, this assumption is very helpful as it provides ample opportunities for optimization. We note also that while it may look very narrow, it actually accurately captures surprisingly many ML applications,
which primarily involve estimating conditional probabilities. Examples include classifiers and regression, feature extraction, different variants of probabilistic graphical models, etc..

Let us consider a set of variables \((Pa(X_i) \cup \{X_i\}) \subseteq \mathcal{X}\) and their corresponding counts \(N_{ij}\) and \(N_{ijk}\), for some specific configuration \(j\) of \(Pa(X_i)\) and \(k\) of \(X_i\). Here we are distinguishing between counts for \((Pa(X_i) \cup \{X_i\})\) and \(Pa(X_i)\) to simplify computing conditional probabilities while maintaining generality – by passing \(Pa(X_i) = \emptyset\) we can execute queries over single variable \(X_i\), and by considering only \(N_{ijk}\) we get joint queries \(Pa(X_i) \cup \{X_i\}\). The key observation is that we can leverage associativity and commutativity, and instead of first gathering all counts and then performing aggregation, we can create a stream of counts corresponding to all unique and relevant configurations found in \(D\), and perform the aggregation directly on the stream. This enables us to push computations to data, mitigating memory overheads due to counts caching. To achieve this, we abstract the computations via a function object (a concept supported by all modern programming languages), which is then repeatedly invoked over the stream. The example function object corresponding to Equation 4.1 is given in Figure 4.2. In the essence, the object receives \(N_{ijk}\) and \(N_{ij}\) via the function call operator (line 3), performs the required intermediate computations, and then aggregates the result into internal state. This internal state can be then inspected (line 7) to retrieve the final result of the aggregation. From the user perspective, the function call operator acts as an interface, and is directly invoked by a routine responsible for enumerating, and emitting, all unique configurations for the variables of interest (see parameter \(F\) in Algorithms 1 and 2 in the following sections). Thus the interface provides a convenient encapsulation, and the end-user who defines the function object (e.g., implementing a scoring function in
BN learning) can focus solely on expressing computations (i.e., high-level logic and correctness), and does not have to worry about potentially complex logic of low-level details (e.g., how counts are enumerated). Additionally, because function object behaves like a function, but has the advantage of possessing an internal state, it is a convenient mechanism to express even the most demanding computations.

While the proposed interface stems from a relatively simple observation, it has several immediate advantages. First, by separating functionality (i.e., data traversing from computations) we gain flexibility to rapidly investigate different data traversal schemes to extract counts, or even alternate between different strategies depending on the query context (e.g., how many variables are involved, their domain, etc.). Second, since counts are aggregated into an isolated state represented by a function object, and multiple objects can coexist independently, multiple groups of queries, each group with individual context, can be executed concurrently and in parallel, e.g., by different threads. Collectively, this makes the proposed design extremely flexible, efficient and easy to use, as we demonstrate in the experimental results section.

4.3.2 Bitmap Strategy

For the specific $X_i$ and $Pa(X_i)$ our task now is to enumerate counts $N_{ij}$ and $N_{ijk}$ for all configurations $j$ and $k$ found in $D$, and then pass the counts to a function object for aggregation. The idea behind the Bitmap strategy is to represent each variable $X_i$ via a set of $r_i$ bitmaps of size $m$, where each bitmap indicates instances for which $X_i$ is in the corresponding state (see Figure 4.1a). Then, the entire process of enumerating counts can be reduced to performing logical
AND on bitmaps, equivalent of set intersection, and to bit counting, equivalent of computing set cardinality. This is summarized in Algorithm 1, with example in Figure 4.1b (for convenience, in the algorithm we use set notation instead of directly representing bitmaps).

**Algorithm 1** \( \text{QUERY}(X_i, Pa, F, b) \)

1: if \(|Pa| = 0\) then
2: \(N_{ij} \leftarrow |b|\)
3: for \(v \in [1, \ldots, r_i]\) do
4: \(b_v \leftarrow \{p \mid D_i[p] = v\}\)
5: \(N_{ijk} \leftarrow |b \cap b_v|\)
6: if \(N_{ijk} > 0\) then
7: \(F(N_{ijk}, N_{ij}) \triangleleft \text{emit new configuration}\)
8: else
9: \(X_h \leftarrow \text{HEAD}(Pa)\)
10: for \(v \in [1, \ldots, r_h]\) do
11: \(b_v \leftarrow \{p \mid D_h[p] = v\}\)
12: if \(|b \cap b_v| > 0\) then
13: \(\text{QUERY}(X_i, \text{TAIL}(Pa), F, b \cap b_v)\)

To execute counting queries for \(X_i\) and \(Pa(X_i)\) (abbreviated to \(Pa\)), and function object \(F\), we perform Depth First Traversal (DFS) over the tree whose leaves represent all possible \(r_i \cdot q_i\) states of interest (recall that \(q_i = \prod_{X_j \in Pa(X_i)} r_j\)). The bottom layer of the tree is induced by the states of \(X_i\), and the top layers correspond to variables in \(Pa\). When moving down the tree (lines 9-13), we compute intersection between the set of instances supporting variables’ configurations seen thus far (in the algorithm denoted by \(b\), which initially consists of all \(m\) instances), and the set of instances supporting current configuration of the considered variable from \(Pa\) (in the algorithm denoted by \(b_v\)). We continue traversal only if the size of the intersection is greater than zero, implying non-zero count for given joint configuration of variables. Once we reach a leaf of the tree (lines 1-7), we compute the final counts \(N_{ijk}\) and \(N_{ij}\) for the corresponding configuration.
tions $j$ and $k$, and emit those via call to $F$.

The depth of the tree depends on the number of variables involved in the query, and the number of leaves is bounded by $O(\min(q_i, m))$, with each step in the traversal involving $O(m)$ cost of computing intersection and cardinality. While the tree could potentially involve exponential (in the number of query variables) number of nodes, it is never explicitly stored in the memory, and even for $D$ with large number of instances many configurations have zero count, allowing for their corresponding sub-trees to be pruned. To further leverage this property, we order $Pa$ such that variables with lowest entropy estimated from $D$ are at the top of the tree. Since variables with low entropy are likely to have configurations for which there will be only few supporting instances, they are more likely to trigger zero counts and hence lead to a smaller tree to traverse. For example, consider executing $\text{Query}(X_2, \{X_1, X_3\})$ outlined in Figure 4.1b. There are total of 7 configurations which we should enumerate, and if we traverse the tree starting from variable $X_3$, which has lower entropy than $X_1$, then we will have to consider 6 intermediate states. If we were to start with variable $X_1$, then this number would increase to 9. This optimization performs extremely well in practice, and, as we show in the experimental results section, for certain ranges of $n$ and $m$, Bitmap outperforms other strategies.

In the practical terms, the strategy can be efficiently implemented using streaming extensions (SIMD) in current processors. Bitmaps for individual variables can be precomputed and laid out in the memory instead of $D$, with acceptable memory overhead (i.e., $m \cdot r_i$ vs. $m \cdot \log_2(r_i)$ bits), and the relative ordering of variables in $D$, based on their entropy, can be established beforehand as well.
4.3.3 Radix Strategy

While the Bitmap strategy is amenable to very efficient implementation, its scalability may still suffer when datasets with very large number of instances are exercised by queries with many variables, or variables with high arity. This is because in such cases the DFS tree will have fewer nodes to prune, and the advantage of fast bit-wise operations will be offset by the poor asymptotic behavior. To address these cases, we consider an alternative approach, which we refer to as Radix strategy. The strategy is derived from the classic radix sort algorithm, and it involves recursively partitioning instances in $D$ such that single partition at given level captures all instances corresponding to one specific configuration of the query variables. This approach is summarized in Algorithms 2 and 3, with example in Figure 4.1c.

**Algorithm 2** \( \text{QUERY}(X_i, Pa, F) \)

1: $B \leftarrow \{1, \ldots, m\}$  
2: if $|Pa| \neq 0$ then  
3: $B \leftarrow \text{BUCKETS}($head($Pa$), tail($Pa$), head($B$))
4: for $b \in B$ do
5: $N_{ij} \leftarrow |b|$
6: if $N_{ij} > 0$ then
7: $B' \leftarrow \text{BUCKETS}(X_i, [], b)$
8: for $b' \in B'$ do
9: $N_{ijk} \leftarrow |b'|$
10: if $N_{ijk} > 0$ then
11: $F(N_{ijk}, N_{ij})$ \( \triangleright \) emit new configuration

The algorithm follows the Most Significant Digit (MSD) radix, with the left most digits being states of individual variables in $Pa$, and the least significant digit representing states of $X_i$ (Algorithm 2, line 3). At each level, the number of newly created partitions is proportional to the arity of the considered variable, and the size of the partition is the support in $D$ for the particular configuration.
Algorithm 3 \textsc{Buckets}(X_p, Pa, b)

1: $B' \leftarrow []$
2: \textbf{for} $q \in [1, \ldots, |b|]$ \textbf{do}
3: \hspace{1em} $x_p \leftarrow D_p[b[q]]$
4: \hspace{1em} $B'[x_p].\text{APPEND}(b[q])$
5: \hspace{1em} \textbf{if} TAIL(Pa) = [] \textbf{then}
6: \hspace{2em} \textbf{return} $B'$
7: \hspace{1em} $B'' \leftarrow []$
8: \hspace{1em} \textbf{for} $b' \in B'$ \textbf{do}
9: \hspace{2em} $B''.\text{APPEND}($\textsc{Buckets}(\textsc{head}(Pa), \textsc{tail}(Pa), b'))$
10: \hspace{1em} \textbf{return} $B''$

The order in which variables from $Pa$ are processed is not significant, since the cost of identifying empty partitions does not induce overheads. Because the actual instances in $D$ are not to be sorted, but only partitioned, it is sufficient that we maintain a list (in algorithms denoted by $B$) of partition descriptors containing indexes of the constituent instances and partition size (Algorithm 3, lines 1-4). As soon as all partitions prescribed by $Pa$ are identified we can proceed to emitting counts (Algorithm 2, lines 4-11), which must be preceded by the final round of partitioning with respect to $X_i$ (Algorithm 2, line 7).

The algorithm requires that for each variable $X_p \in Pa$ its corresponding data vector $D_p$ is completely scanned, leading to the overall $O(|Pa| \cdot m)$ complexity. In practice, the entire method is efficiently implemented by first organizing the database $D$ in the column-major format, and then maintaining a FIFO queue of partition descriptors, with $O(m)$ auxiliary space to keep track of the assignment of indexes to partitions. Moreover, partitioning for individual variables can be precomputed in advance, further bootstrapping the first step of the algorithm.

To conclude the presentation, we note that both Bitmap and Radix strategies can be further augmented such that instead of enumerating all counts (i.e., executing queries with shared context) they deliver counts just for the specific
assignment of the query variables. To achieve this, it is sufficient to process only a single path from the root to the leaf with the target assignment in the DFS tree of the Bitmap strategy, and to find the partition corresponding to the assignment, instead of all partitions, in Radix.

4.4 Experimental validation

We implemented both proposed strategies as a C++ software library, which we complemented with Python bindings for the ease of use. At its core, the library uses standard SSE SIMD intrinsics to implement basic bitmap operations (i.e., logical \texttt{AND}, and bit counting), and it exposes all functionality via the interface described in Section 4.3.1. The resulting open source package, which we call \texttt{SABNAtk}, is available from: \url{https://gitlab.com/SCoRe-Group/SABNAtk}.

We deployed \texttt{SABNAtk} on a server with two Intel Xeon E5-2650 2.30 GHz 10-core CPUs, and 64 GB of RAM. To test the performance, we ran a series of experiments using popular ML benchmark datasets (see data summary in Table 4.1). For reference, we used hash table from the C++ standard library, and the sparse ADtree data index [72]. The hash table represented contingency table created by directly scanning the input database, with keys encoding specific assignment of variables in $Pa(X_i)$, and values representing vectors of counts for specific assignment of $X_i$. To maximize cache memory usage, the strategy operated on the database stored in the row-major order. Finally, to make the comparison fair and avoid biases due to the differences in programming languages, we developed an efficient ADtree implementation in C++. We note that other available implementations, for example [78], turned out to be substantially slower than our version.
Table 4.1: Benchmark data used in experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Insur</th>
<th>Mild</th>
<th>Alarm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child</td>
<td>20</td>
<td>2-6</td>
<td>3</td>
</tr>
<tr>
<td>Insurance</td>
<td>27</td>
<td>2-5</td>
<td>3.3</td>
</tr>
<tr>
<td>Mildew</td>
<td>35</td>
<td>3-99</td>
<td>16.4</td>
</tr>
<tr>
<td>Alarm</td>
<td>37</td>
<td>2-4</td>
<td>2.83</td>
</tr>
<tr>
<td>Barley</td>
<td>46</td>
<td>3-67</td>
<td>9.02</td>
</tr>
<tr>
<td>Hailfinder</td>
<td>56</td>
<td>2-11</td>
<td>3.98</td>
</tr>
<tr>
<td>Win95(pts)</td>
<td>74</td>
<td>2-2</td>
<td>2</td>
</tr>
<tr>
<td>Pathfinder</td>
<td>104</td>
<td>2-63</td>
<td>4.2</td>
</tr>
</tbody>
</table>

In the following, we discuss in detail several key results obtained using the above setup. More extensive results (including additional test cases), together with the data that can be used to reproduce our experiments, are available from: https://gitlab.com/SCoRe-Group/SABNAAtk-Benchmarks.

Before we proceed with the results discussion, we note that in order to use ADtree, the input database has to be first indexed. In all our experiments, we considered only the query time with index already loaded into memory. Moreover, ADtree provides a hyper-parameter $\ell$ to configure the size of the leaf-lists [72]. We experimented with several values of the parameter, to fine-tune the trade-off between query performance and memory consumption, and we settled with $\ell = 16$, which we use throughout the chapter. The results obtained for other ADtree configurations exhibited similar patterns to those reported in the paper, and are available online.

### 4.4.1 Queries in Bayesian Network Structure Learning (BNSL)

Counting queries with shared context are the key operations performed in score-based BNSL and Markov blankets discovery [1]. Both problems depend on the parent set assignment as a sub-routine [16], and for given $X_i$ can be solved exactly by traversing a lattice with $n$ levels formed by the partial order set inclu-
Figure 4.3: The total execution time of the parent set assignment solver with ADT, Hash, BMap and Rad strategies, normalized with respect to the fastest method. The solver was executed up to the level where $|Pa| = 6$.

**Problem Statement**

Given $\mathcal{X} - \{X_i\}$. For given $\mathcal{X}$ and $D$, queries of the form $\text{Query}(X_i, Pa, F)$ are performed for each $X_i$, where $Pa$ iterates over all possible subsets of $\mathcal{X} - \{X_i\}$, starting from empty set. Hence, at level $i = 0, \ldots, n - 1$ we have that $|Pa| = i$, and there are total $\binom{n-1}{i}$ queries to execute, creating interesting pattern of queries that grow in size as computations progress. The function object $F$ implements decomposable scoring function, e.g., MDL \[43\], BDeu \[44\], etc., that evaluates the assignment of $Pa$ as parents of $X_i$. Chapter 5 discusses parent set assignment problem in detail.

We used all tested strategies to implement counting queries in the open source parent set assignment solver \[79\]. The solver uses MDL scoring function, deploys several optimizations to eliminate some of the queries based on the results seen thus far, and because it effectively explores large combinatorial search space it has significant memory requirements. It can also leverage Intel TBB tasks to execute multiple queries in parallel. As such, it serves as a practical benchmark for the query strategies. In our experiments, instead of considering all possible parent set sizes, as required by the exact solver, we limited the solver to $|Pa| \leq 6$, to make tests computationally feasible. This corresponds to a heuristic in which we make an assumption that no variable in the final BN can have more than six parents.
Figure 4.3 shows the total execution time of the solver for different input databases and query strategies. From the plots, we can see that our proposed strategies significantly outperform ADtree and HashTable, across all benchmarks. In fact, for datasets with high-arity variables, i.e., Mildew and Barley, the Radix strategy is 100 times faster than ADtree. This is explained by very large number of states that are to be expected in such datasets (and are costly to manage by ADtree), and by the pattern of how queries are generated by the solver. Because the size of the queries and their number grow together, there are only a few small queries that benefit ADtree, and increasing number of queries that are easily handled by the Radix strategy.

To illustrate how critical is the performance of counting queries for parent sets assignment, in Table 4.2 we report the total execution time of the solver, together with the fraction of the time taken by the queries, when running on databases with 100K instances. In all cases, the execution is dominated by database querying that accounts for 90%-99% of the total time. Interestingly, this fraction is smaller for ADtree than for other strategies, even though ADtree is slower (we observed this pattern in all test cases). We believe that this is because BMap and Rad are memory friendly, and have minimal effect on memory utilization by the solver, thus minimizing cache update overheads, which in turn could slow down the solver. This is not the case for ADtree, which requires gigabytes of memory to run, and hence influences performance of the solver, affecting the ratio between the query and the solver time.
Table 4.2: Execution time of the parent set assignment.

<table>
<thead>
<tr>
<th></th>
<th>Insur</th>
<th>Mild</th>
<th>Alarm</th>
<th>Barley</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>35m20s</td>
<td>–</td>
<td>119m</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>90.2%</td>
<td>–</td>
<td>98.2%</td>
<td>–</td>
</tr>
<tr>
<td>Hash</td>
<td>26m26s</td>
<td>92m3s</td>
<td>190m47s</td>
<td>276m16s</td>
</tr>
<tr>
<td></td>
<td>98.2%</td>
<td>99.1%</td>
<td>99.4%</td>
<td>98.2%</td>
</tr>
<tr>
<td>BMap</td>
<td>17m28s</td>
<td>1107m57s</td>
<td>100m55s</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>98.8%</td>
<td>99.8%</td>
<td>99.9%</td>
<td>–</td>
</tr>
<tr>
<td>Rad</td>
<td>4m41s</td>
<td>37m28s</td>
<td>43m4s</td>
<td>156m32s</td>
</tr>
<tr>
<td></td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

4.5 Related work

As we mentioned throughout the chapter, counting queries in machine learning applications are often handled via some variant of the ADtree data index. The sparse ADtree [72, 75], which we used in our experiments, precomputes and caches counts for all possible variable configurations. The counts are organized into a tree of *vary nodes*, encoding the choice of variables to facilitate fast searching, and *AD nodes* that store the actual query counts. To partially mitigate the excessive memory use, ADtrees do not explicitly represent most commonly occurring counts, and instead of creating AD nodes for counts lower than certain threshold, they resort to on-demand counting when such nodes are accessed. These base ideas have been extended by multiple researchers to account for dynamic data (i.e., updates to the database) [80], and to improve performance on high-arity data [76, 77]. However, as the core functionality in these data structures remains exactly the same, they suffer from the same limitations that we demonstrated in our experiments (expensive preprocessing, large memory footprint, significant traversing overheads).

Support for counting queries is a primary component in any database management system. In such systems, the query mechanism must support conjunctive queries over multiple tables, and with a variety of possible query predic-
cates [46]. Moreover, the queries are typically executed over tables that cannot be fully materialized in the main memory. Our Bitmap strategy can be viewed as a practical realization of the Leapfrog Trie Join [81] with an unary relation, under assumption that the entire database resides in the main memory.

The idea of using bitmaps to represent sets and their operations (e.g., intersection, cardinality, etc.) is frequent in software and databases design. This is because it allows to reduce memory, storage or network bandwidth, while maintaining the basic sets functionality [82]. In these applications, bitmaps are typically compressed following methods like for example RLE encoding or Roaring [83, 84]. The compressed bitmaps are orthogonal to our approach, and in fact we could use them to improve memory profile of our Bitmap strategy. However, as the compression induces computational overheads, and the size of the databases we consider practical is relatively small, currently we do not use compression.

## 4.6 Conclusions

In this paper, we describe efficient strategies for handling counting queries in machine learning applications. By combining convenient programming interface with memory efficient data traversing algorithms we are able to scale to large data instances, which we confirm via extensive experiments. The proposed solutions outperform and can substitute popular ADtree index. Moreover, to maintain best possible performance across different data instances, they can be selectively applied at the runtime depending on the properties of the queries.

While our approach is presented as a method for static databases, we note
that it can be easily adopted to the cases where the input database expands with new instances during processing. This would amount to a simple update to the bitmaps in the Bitmap strategy, and is automatically handled in the Radix strategy.
Overview. In Machine Learning, the parent set identification problem is to find a set of random variables that best explain selected variable given the data and some predefined scoring function. This problem is a critical component to structure learning of BNs and Markov blankets discovery.

In this chapter, we introduce a new distributed memory approach to the exact parent sets assignment problem. To achieve scalability, we derive theoretical bounds to constraint the search space when MDL scoring function is used, and we reorganize the underlying dynamic programming such that the computational density is increased and fine-grain synchronization is eliminated. We then design efficient realization of our approach in the Apache Spark platform. Through experimental results, we demonstrate that the method maintains strong scalability on a 500-core standalone Spark cluster, and it can be used to efficiently process data sets with 70 variables, far beyond the reach of the currently available solutions.
5.1 Introduction

In Machine Learning, the parent set assignment problem is to find a set of random variables that best explain a selected variable given input data and some predefined scoring criterion [16]. It is a prerequisite to the structure search routine in the BNSL pipeline, where it is solved for each variable to produce a list of potential predecessors of that variable in a final network, which reduces the number of structures that have to be considered [54, 29, 58]. It is also closely related to the feature selection problem, since it directly translates into Markov blankets discovery [85]. Because of these connections, the problem has many practical applications spanning clinical decision support systems, risk assessment, strategic planning, fraud detection and many others [86, 87, 88, 89]. In all these applications, random variables model attributes of interest, their realizations are observed from data, and a model obtained by solving the parent set assignment provides insights into how different attributes depend on each other, including conditional dependencies.

While the parent set identification is critical to building models like BNs, it is known to be formally hard for the most commonly used scoring functions. For instance, it is $\mathcal{NP}$-complete for the Normalized Maximum Likelihood (NML) criterion [16]. Consequently, the current approaches, which we briefly review in Section 5.5, either depend on heuristics or deliver exact solutions but are limited in how large instances they can solve. In fact, the largest problems solved by exact algorithms do not contain more than 40 variables [28]. In contrast, the real-world systems that strongly depend on the high quality BNs often involve hundreds of variables. The available heuristics that can solve instances of that size, e.g. [29], do not provide any guarantees on the quality of the solutions they
find. This significantly impacts their usefulness, since the inherent uncertainty of the model due to the input data and the scoring function, cannot be separated from the deficiencies of the learning algorithm [1]. Consequently, there is a gap between the quality and the size of the models that depend on the exact parent set identification and that can be efficiently learned from the data.

Responding to the above challenges, in this chapter, we propose a new distributed memory algorithm for the exact parent sets identification problem. Our goal is to push the limit on the scale of instances that can be solved in acceptable time limits on a modestly sized parallel cluster. To this end, we make the following specific contributions: 1) we propose a new strategy to constrain and reorganize dynamic programming computations in the parent set assignment problem such that computational grain is improved and fine-grained synchronization is avoided, 2) we define a simple mechanism that we use to change the mode of computations from BFS to DFS such that the main memory is preserved. To validate our approach, we provide an efficient implementation on the Apache Spark platform [90], and demonstrate its strong scalability across different ML test sets. We then show that on a 500-core cluster with 25 nodes, our system can process HEPAR II test data [91] in slightly over 20 hours. With 70 variables and small variability, this data set is one of the most challenging benchmarks for BNSL, and it has no exact results available to date.

This chapter is organized as following: in Section 5.2, we formally introduce how parent set assignment solver is used as a subroutine in BNSL. In Section 5.3, we introduce our proposed method, and we describe its experimental validation in Section 5.4. We summarize related work in Section 5.5, and conclude the paper in Section 5.6.
5.2 Bayesian Network Structure Learning (BNSL)

In the context of BNSL, it is necessary to consider a slightly broader version of the parent set assignment problem.

We will say that $U \subseteq \mathcal{X} - \{X_i\}$ is a maximal parent set of $X_i$ if $d(X_i, U) = s(X_i, U)$. From Equation 2.1 we have that if $U$ is a maximal parent set then no subset of $U$ has score better than $d(X_i, U)$, i.e. $\forall U' \subset U d(X_i, U) < d(X_i, U')$. Hence, by identifying all maximal parent sets of $X_i$ and memoizing their corresponding scores $s$, we can efficiently answer queries about any optimal parent set of $X_i$. Specifically, to answer query $d(X_i, U')$ for any $U'$ it is sufficient to check if $U'$ is one of the maximal parent sets of $X_i$. If it is, then all we have to do is to return the memoized score $s$ of that maximal parent set. Otherwise, $d(X_i, U')$ must be equal to the smallest $s$ among all maximal parent sets of $X_i$ for which $U'$ is a superset.

The above property of maximal parent sets has important practical implications. For example, to compute the score $Q(\mathcal{X})$ of an optimal BN over $\mathcal{X}$, and thus find the network itself, we have to solve recursion 6.1 for $Q^*(\mathcal{X})$. Even with efficient algorithms such as [54, 58] this requires large and hard to predict number of queries for optimal parent sets, owing to the component $d(X_i, U - \{X_i\})$ in the recursion. As for a single variable $X_i$ there are $2^{n-1}$ optimal parent sets, memoizing them all is impractical and often infeasible. In contrast, the set of all maximal parent sets is usually many orders of magnitude smaller as reported in Table 6.1, and hence using it instead, in the way we explained before, is the desired and viable alternative [54, 58, 29].

From the computational point of view, identifying maximal parent sets of $X_i$ is the same as selecting its optimal parent set from $\mathcal{X} - \{X_i\}$. The only differ-
ence is extra steps required to test and store subsets that correspond to maximal parent sets. In practical settings, we wish to enumerate all maximal parent sets for all variables in $\mathcal{X}$.

### 5.3 Proposed Approach

![Diagram](image.png)

Figure 5.1: (a) Example of the dynamic programming lattices for $\mathcal{X} = \{X_1, X_2, X_3, X_4\}$. Processing node $U$ in a lattice for variable $X_i$ requires computing $s(X_i, U)$ and access to $d(X_i, U')$ for each predecessor $U'$ of $U$. (b) Example constrained lattices, and (c) their corresponding “folded” representation. A node $U$ in the compacted lattice requires that $s$ is evaluated for several variables that share candidate parents $U$. This improves efficiency of computing $s$, decreases memory requirements and increases computational density. (d) Precedence constraints after eliminating fine-grain synchronization within every layer of the lattice in (c). Nodes processed by the same task if the DFS mode is initiated at layer $l = 1$ are outlined. Note that following how input variables are ordered, the nodes in the larger tasks are more likely to be pruned.

Given a set of variables $\mathcal{X}$, database of observations $D$, and a scoring function $s$, our goal is to enumerate all maximal parent sets for all $X_i \in \mathcal{X}$. If we con-
sider a single variable $X_i$, then we can directly apply recursion in Equation 2.1
and starting from the empty set we can consider parent sets of growing size.
This process can be thought of as a top-down traversal of the dynamic program-
ing lattice with $n$ levels formed by the partial order “set inclusion” on the
power set of $\mathcal{X} - \{X_i\}$ (see Figure 5.1a). At the level $l = 0$ of the lattice we have
empty set. Two nodes in the lattice, $U'$ and $U$, are connected only if $U' \subset U$
and $|U| = |U'| + 1$. Here we use $U$ to denote both a subset of $\mathcal{X} - \{X_i\}$ and
the corresponding node in the lattice. A node $U$ represents a parent set of $X_i$.
When it is discovered, we compute $s(X_i, U)$, compare it with scores $d$
passed by its predecessors to obtain $d(X_i, U)$, and if $U$ is a maximal parent set we store or
report a tuple $(X_i, U, s(X_i, U))$.

While the above strategy is clearly guaranteed to enumerate all maximal
parent sets, it is both computationally and memory challenging. The computa-
tional complexity is due to the $\Theta(2^n)$ invocations of $s$, and memory complexity
is driven by how the dynamic programming lattice is traversed. For example,
one way is to assume BFS traversal induced by the precedence constraints in the
lattice. In such case, maximal parent sets are enumerated layer-by-layer with a
synchronization point between any two consecutive layers. This strategy re-
quires that both layers are stored in the memory, which implies $O\left(\left(\frac{n}{2}\right)^2\right)$ space
complexity, irrespective of which parallel BFS realization we assume [92]. An-
other approach is to use some variant of DFS. With DFS we can benefit from
techniques like hypercube pipelining, similar to [60], but this strategy requires
that we store partial results and update them each time a node is discovered
before its all predecessors are processed. As a result, the space complexity is
$O(2^n)$ and we have to maintain potentially irregular memory updates to detect
new maximal parent sets. Finally, while all variables in $\mathcal{X}$ can be processed in-
dependently, the resulting embarrassing parallelism is highly limited. This is because the computational cost for a single variable is exponential in $n$, which effectively constraints the total number of variables we may hope to process. For example, if we assume $n = 48$ then the estimated memory requirements to process a single lattice, with $O(2^n)$ nodes and 16 B per node, is 4 PB with a modest 48-way parallelism.

### 5.3.1 Constraining the Search Space

To achieve a scalable strategy, we start from constraining the search space. This is necessary since the exponential cost of considering all optimal parent sets is prohibitive for realistic problem instances, irrespective of how efficient is our parallel exploration algorithm.

For every variable $X_i$ it is reasonable to expect that its optimal parent set will not contain all other variables. In other words, there is a limit on the depth to which we should be exploring the dynamic programming lattice of $X_i$. To maintain exactness guarantees, we have to ensure that the bound on the depth of exploration is no smaller than the unknown size of the optimal parent set. Here we provide such a bound for the information-theoretic MDL scoring function. We note that similar bounds can be derived for other functions, and in fact a significant work in this direction has been done, for example in [26].

The MDL score is defined as:

$$s(X_i, U) = m \cdot H(X_i|U) + NC(X_i, U),$$

(5.1)

where

$$NC(X_i, U) = \frac{1}{2} \cdot \log_2(m) \cdot q_i \cdot (r_i - 1)$$

(5.2)
is a network complexity term, $\mathcal{H}(X_i|U)$ is the estimated conditional entropy of $X_i$ given $U$, and $r_i > 1$ is the number of states of $X_i$, and $q_i = \prod_{j: X_j \in U} r_j$ is the number of states that variables in $U$ can assume ($q_i = 1$ if $U = \emptyset$). The parameters $r_i$ and $q_i$ as well as the conditional entropy are directly assessed from $D$. In short, the MDL score of a pair $(X_i, U)$ is the number of bits required to encode information about $X_i$ and its parents $U$ if we were to use Huffman coding of $D$.

To derive the bound we exploit the following observation. When $U$ is empty, we have the maximal conditional entropy $\mathcal{H}(X_i|\emptyset) = \mathcal{H}(X_i)$ and the minimal network complexity $\mathcal{NC}(X_i, \emptyset) = 0.5 \cdot \log_2(m) \cdot (r_i - 1)$. By increasing the size of $U$ we can decrease conditional entropy of $X_i$, which has the theoretical limit of 0, at the expense of increasing network complexity. This follows from the basic properties of entropy and the definition of the network complexity term. Once the network complexity $\mathcal{NC}(X_i, U)$ is greater than or equal to $s(X_i, \emptyset) = m \cdot \mathcal{H}(X_i) + \mathcal{NC}(X_i, \emptyset)$, irrespective of which variables we add to $U$, the score $s(X_i, U)$ will always increase. This is the point at which network complexity outweighs any gains from the decreasing entropy of $X_i$. Consequently, if $U$ satisfies

$$\text{Condition 1: } \mathcal{NC}(X_i, U) \geq s(X_i, \emptyset),$$

then any superset of $U$ can be excluded from further consideration, since it will not admit new optimal or maximal parent sets for $X_i$. The efficiency of Condition 1 depends on the input data $D$. Nevertheless, it works extremely well in practice. For example, in our experiments, reported in Section 5.4, we found that for real-world data with $n = 70$ we never considered nodes with more than nine variables.
We can further extend our pruning strategy by using the following observation [93]. The lowest entropy we can achieve for $X_i$ is $H(X_i, \mathcal{X} - \{X_i\})$. Now consider the score $d(X_i, U)$. Here we have that if

$$\text{Condition 2: } d(X_i, U) \leq m \cdot H(X_i, \mathcal{X} - \{X_i\}) + N\mathcal{C}(X_i, U)$$

holds, then no superset of $U$ can improve the score $d(X_i, U)$. This is because any superset of $U$ has higher network complexity, and hence $\forall U \supset U' m \cdot H(X_i, \mathcal{X} - \{X_i\}) + N\mathcal{C}(X_i, U) \leq H(X_i, \mathcal{X} - \{X_i\}) + N\mathcal{C}(X_i, U')$. As previously stated, if $U$ satisfies the condition we can exclude it from further considerations, since it will not admit new optimal or maximal parent sets. The example effect of applying our pruning conditions is shown in Figure 5.1b.

Although both conditions achieve the same goal of pruning the search space, they differ in which information they require. To test Condition 1, we use only network complexity, which can be computed for any pair $X_i$ and $U$ independently of other $U' \subset U$, i.e. independently of predecessors of $U$ in the dynamic programming lattice. On the other hand, Condition 2 provides a tighter bound but depends on the score $d(X_i, U)$, which, as we explained earlier, requires access to the maximal parent sets of $X_i$.

### 5.3.2 Parallel Exploration

Because of the memory and computational complexity, which remains challenging even when our pruning conditions are applied, we focus our parallel strategy on the distributed memory systems, with the Apache Spark platform [90] serving as an execution vehicle.

Recall that our goal is to traverse in the top-down fashion the dynamic pro-
gramming lattices for all $X_i$. A node $U$ in a lattice corresponds to a computational task that evaluates $s(X_i, U)$, tests if $U$ is a maximal parent set, and checks if supersets of $U$ can admit new maximal parent sets. These tests are the source of precedence constraints between the tasks. The main idea of our parallel approach is as follows. We on-the-fly generate and “fold” the dynamic programming lattices for different $X_i$ into a single lattice with lower memory requirement and denser computational load. We explore the resulting lattice in parallel, initially in the BFS mode, and switch to DFS when memory becomes a bottleneck. To store and access maximal parent sets discovered in the process, we maintain a global state, which is synchronized via reduction between the layers. Finally, we reorder computations within each layer to eliminate fine-grain synchronization between the tasks, that otherwise would be necessary to effectuate our pruning conditions. Below, we explain each element of our approach.

5.3.2.1 Folding Lattices

If we consider the dynamic programming lattice for variable $X_i$, then until our pruning conditions become effective we have to manage \( \binom{n-1}{l} \) tasks at the level $l$ of the lattice. Consequently, the memory required to represent the entire layer $l$ is bounded by $B_1 = c_1 \cdot n \cdot \binom{n-1}{l}$, assuming cost $c_1$ to store a task. This easily becomes problematic for larger problems as soon as $l > 2$. The problem persists even when pruning takes place, since initially only some of the tasks are removed from consideration. However, we can “fold” the dynamic programming lattices such that the tasks sharing the same set $U$ across different lattices are represented by a single task (see Figure 5.1c). Let the memory taken by such combined task be $c_2$. The memory requirement of the new lattice is $B_2 = c_2 \cdot \binom{n}{l}$.
This gives us $\frac{B_1}{B_2} = \frac{c_1}{c_2} \cdot (n - l)$ reduction in memory complexity. To store a task we can use a bitmap, where $i$-th bit indicates whether element $i$ is in a set. In such case, $c_2 = 2 \cdot c_1$, since in the compacted lattice we require one bitmap to represent all $X_i$ for a task, and one bitmap to represent the actual $U$ (v.s. storing only $U$ in the original lattice). By using bitmaps we additionally reduce memory overhead, and we can realize basic operations, like testing set inclusion, with only few hardware instructions. The memory reduction becomes less significant as $l$ increases. However, this is acceptable, since we expect that, thanks to the pruning conditions the search process will terminate early, which we confirm via experimental results.

The main advantage of our “folding” step is significantly increased computational density. To process a single task in the “folded” lattice, we have to perform multiple evaluations of $s$ with the same parent set $U$. We note that by having the same parents in the consecutive invocations of $s$, we can precompute statistics about $D$ induced by $U$, and reuse them from one invocation to another. Consequently, the cost of processing a task in the “folded” lattice is higher than the cost of processing an individual corresponding task in the original lattice, but it is lower than the total cost of processing all corresponding tasks from the original lattices, i.e. if $X$ is a set of random variables sharing $U$ we have that $T(s(X, U)) < \sum_{X_i \in X} T(s(X_i, U))$, where $T$ is the processing cost.

By “folding” the lattices, we limit parallelism in the first two stages of the lattice. However, this has a negligible effect on the scalability, since even for large $n$ the cost of processing these layers is minimal compared to the total processing time. Alternatively, we may decide to “fold” the lattices only after the desired level of parallelism is achieved. Finally, the computational cost of individual
tasks becomes non-uniform, but this is addressed by the dynamic scheduling at the run-time.

### 5.3.2.2 Limiting Synchronization

Consider the task for node $U_i$ at the layer $l$, and suppose that *Condition 1* or *Condition 2* holds for $U_i$. In such case, no task that corresponds to a superset of $U_i$ should be generated and included in the layer $l + 1$, as it will not contribute new maximal parent sets. In other words, at given layer we should see only those tasks whose predecessors all did not satisfy the pruning conditions. However, to enforce this requirement we would need either complex synchronization between all tasks within the same layer, or a reduction operation on all possible tasks for the next layer, which effectively would defy the purpose of pruning.

To address this problem, we can change the way in which tasks for the next layer are enumerated, such that synchronization is bypassed at the small cost of considering a few unnecessary tasks in the next layer. We first order variables in $\mathcal{X}$ by the decreasing number of states they have in $D$, i.e. for any $X_i$ and $X_j$, if $i < j$ then $r_i \geq r_j$, and we maintain this ordering for every node $U_i$. If two variables have the same number of states, we use $\mathcal{H}(X_i, \mathcal{X} - \{X_i\}) < \mathcal{H}(X_j, \mathcal{X} - \{X_j\})$ as a secondary condition. Then, when deciding whether a task should be considered in the next layer, instead of checking if any of its predecessors satisfied pruning condition, which would require synchronization, we check only one selected predecessor. Specifically, let $X_j \in U_i$ be the maximal element in $U_i$. To enumerate descendants of $U_i$, we consider only $U'_i = U_i \cup \{X_k\}$ for all $k > j$. Thus, node $U_i$ becomes a predecessor to $n - j$ nodes (see Figure 5.1d). At the same time, from Equation 5.2 and *Conditions 1* and *2*, it follows that smaller the $j$ the higher the probability that $U_i$ will satisfy pruning conditions. To see
why, observe that the network complexity term grows as the product of the number of states that variables in $U$ can assume. Because variables are ordered by the decreasing number of their states and the increasing entropy, we have that if $|U| = |U''|$ and $j < j''$, where $X_j$ is the maximal element in $U$ and $X_j''$ is the maximal element in $U''$, then $\mathcal{NC}(X_i, U) \geq \mathcal{NC}(X_i, U'')$. Consequently, nodes that are predecessors to the largest number of nodes in the next layer are most likely to meet the pruning conditions. While this approach does not guarantee that all tasks that should be pruned will not be generated, it works very well in practice. In fact, in our experiments we found that we remove no less than 97% of all tasks that should be pruned. The remaining 3% constitute an extra work of processing nodes that do not contribute maximal parent sets. Note that these nodes once processed never create successors and thus the extra work overhead does not propagate.

To decide whether node $U$ at layer $l$ is a maximal parent set for $X_i$, we require optimal parent set scores, $d(X_i, U')$, for all subsets $U' \subset U$ from the layer $l - 1$. As we explained earlier, instead of maintaining a complete list of all optimal parent sets, to retrieve $d(X_i, U)$ we depend only on the previously enumerated maximal parent sets. For each variable $X_i$, we store a list $L(X_i)$ of its maximal parent sets represented by tuples $(U, s(X_i, U))$, and sorted by the score $s(X_i, U)$. Then to extract all optimal parent set scores for $X_i$ and $U$ we require $O(|L(X_i)|)$ scan of $L(X_i)$. This is affordable, since even complete $L(X_i)$ is very small for a typical input data (see Table 5.1 in Section 5.4). Each task at layer $l$ may contribute a new maximal parent set that must be available to all tasks for $X_i$ in the subsequent layers. Consequently, we maintain all $L(X_i)$ as a global state that is updated via all-to-all reduction, with list merging as an operator, after given layer is entirely processed. This step can be efficiently executed considering a
small size of $L(X_i)$.

5.3.2.3 Changing Exploration Mode

While our pruning conditions significantly constrain the search space, for large problems the number of the tasks generated in the later stages of the execution may still exceed the available main memory. This in turn would lead to the undesired out-of-core execution. After processing all tasks at layer $l$, we can estimate the number of tasks that layer $l + 2$ will have in the worst case. If that number exceeds the total available memory, it is reasonable to conclude that we have sufficient parallelism, and instead of creating new tasks we can change the mode of execution into a memory preserving DFS. Specifically, for each node $U$ at layer $l + 1$, instead of considering all supersets of $U$ independently, we can process them sequentially following the DFS order (see Figure 5.1d). However, in such case we cannot assume that the global list of all maximal parent sets is consistent between different tasks. As a result, some tasks could end up generating incorrect maximal parent sets or could perform extra work because without the access to the complete list of maximal parent sets Condition 2 could unnecessarily fail. To mitigate this situation, we flag all maximal parent sets generated in the DFS mode that potentially could not be maximal in the global sense, i.e. when maximal parent sets from other tasks are taken into the account. These are maximal parent sets $U$ whose at least one strict subset $U' \subset U$ has been processed in a different task. Once all tasks are processed, we perform reduction to obtain the final global state for all $L(X_i)$. Then, we proceed with checking if the flagged maximal parent sets remain maximal in the merged $L(X_i)$. Let $z_f(X_i)$ be the total number of maximal parent sets flagged when running in the DFS mode. The cost of verifying these maximal parent sets is $O(z_f(X_i) \cdot |L(x_i)|)$. 
This is because, in the worst case, for every flagged maximal parent set $U$, we have to check if none of the remaining elements in $L(X_i)$ is a subset of $U$ with a better score. However, it turns out that in the practical settings $z_f(X_i)$ is a very small number, and in fact frequently we have that $z_f(X_i) = 0$. To understand why, consider the following. The memory requirements due to BFS grow exponentially with the depth of the dynamic programming lattice. At the same time, because the network complexity term, $NC$, grows exponentially as well, the probability of enumerating new maximal parent sets decreases as we progress to the higher layers of the lattice. In our experiments, for all tested data sets we did not enumerate new maximal parent sets beyond layer $l = 6$. At the same time, if the available main memory is limited, and we are forced to switch to the DFS mode early, then we can expect that the majority of the maximal parent sets tested by a single task will not be depending on the maximal parent sets discovered in other tasks. This is a direct consequence of the precedence constraints within the lattice.

When switching to the DFS mode, we can expect an increased computational imbalance between the tasks. However, the largest tasks which could be the source of the most significant imbalance are the ones which are the most likely to be pruned. At the same time, the number of the DFS tasks will remain sufficient to provide room for load balancing at the run-time, which we confirm by experiments.

5.3.3 Apache Spark Implementation

We implemented our parallel approach using the Apache Spark platform. The reason we use Spark is purely pragmatic: the platform supports locality-aware
dynamic task scheduling, which we directly benefit from, since our computational tasks can be heterogeneous owing to the lattice “folding” and the potential use of the DFS mode. Additionally, Spark API makes expressing iterative BSP-style programs extremely productive. While it is clear that using one of the traditional HPC models, e.g. MPI or UPC, we could probably achieve faster implementation, we believe that the scalability would remain comparable.

The high level exploration components of our method are implemented in Python, and the computationally intensive parts, specifically evaluations of function \( s \), are offloaded to the efficient, SIMD-parallel C++ kernel derived from our SABNA package [58, 79]. Apache Spark is usually regarded as a platform for the data intensive computing. In our case, the input data is typically very small (i.e. at the order of MB), however, it quickly generates massive new data representing individual tasks of the dynamic programming lattice. Below we provide details of the implementation assuming that reader is familiar with the basics of the Apache Spark interface [94].

We follow the standard BSP model realized via iterative transformations on a sequence of Spark Resilient Distributed Datasets (RDDs), where \( RDD_i \) represents layer \( i \) of the compacted dynamic programming lattice. To represent a node in the lattice, RDD stores a tuple \( (X, U) \), where both \( X \) and \( U \) are expressed via bitmaps, and \( X \) keeps variables that share \( U \). To obtain \( RDD_i \), we initialize and parallelize \( RDD_0 \) on Spark driver, since this is very inexpensive operation. Then, we iteratively apply the following transformations: \( RDD_i = RDD_{i-1}.repartition(p).mapPartitions(M) \), where \( p \) is a small multiple, usually four (as suggested in several Apache Spark best practices), of the total number of cores that Spark executors can use, and \( M \) is the mapping function that: 1) evaluates function \( s \) for all variables in \( X \), and identifies potential maximal
parent sets, 2) checks the conditions to constraint the search space, and 3) accordingly yields nodes for the next layer to explore. The repartitioning transformation is to ensure good load balancing between executors since the number of tasks grows from one RDD to the next. Here we depend on the default Spark scheduler. We use mapPartitions, instead of a more natural map, to enable indexing of the data \( D \) when map \( M \) is initialized. By indexing \( D \) we significantly accelerate computations of \( s \), and by doing so only once per partition we avoid unnecessary overheads. At the end of every iteration we materialize RDD by invoking Spark’s count action. Based on the resulting size of RDD, we assess the memory requirement for the subsequent iterations, and decide whether we should be switching to the DFS mode. Finally, at any point of the execution we make sure that the last two RDDs are cached and remain in the main memory to avoid expensive RDD recomputing or restoring from the secondary storage.

The mapping function \( M \) makes use of the information about maximal parent sets from previous iterations, i.e. \( L(X_i) \) for all \( X_i \). To maintain this global state we use a combination of Spark accumulator and broadcast variables, since the memory cost of representing maximal parent sets is very small. In a given iteration, newly discovered maximal parent sets are added by each executor to a customized Spark accumulator to form an update to the global list of all maximal parent sets. As this could lead to potential duplicate entries in \( L(X_i) \) when a task fails or speculative execution is enabled, we make sure that only unique entries are considered. After the count action at the end of the iteration is performed, the accumulator is reduced and the global list managed by the Spark driver is updated and broadcast back to the executors. Together with the count and repartition step, these operations represent communication and synchronization stages in the BSP model.
In the DFS mode, instead of generating RDD for the next layer, which would exceed the available main memory, we apply another transformation to the current RDD, where we explore each partition as described in Section 5.3.2.3. As we explained earlier, this increases the computational cost of every task and makes tasks more heterogeneous. However, at this stage the number of RDD partitions and the distribution of their computational load is such that the Spark run-time can easily maintain load balance.

5.4 Experimental Results

To understand performance characteristics of our approach, we performed a set of experiments on a standalone Apache Spark cluster with 25 nodes and GbE interconnect. Each node in the cluster is equipped with 20-core dual-socket Intel Xeon E5v3 2.3 GHz processor, 64 GB of RAM and a standard SATA hard drive. The shared file system is run under GPFS, however, this is of minor importance considering that the input data is very small, even for the largest considered problems, and it is accessed only once at the very beginning of the computations. In all tests, Spark driver runs with the default parameters, and since it is a very light-weight process, it is collocated with one of the executors. We allocate one Spark executor per node using the default configuration for the number of cores, i.e. each executor uses all available cores in a node. All executors are configured to use 58GB out of the available 64GB, with the remaining memory available to the operating system and Python interpreter. We note that we tested different configurations of executor-to-core ratios, across different data sets, without noticeable difference in performance.

We used several popular benchmark data sets from the UCI Machine Learn-
Table 5.1: Data sets used in experiments.

| Name     | $n$ | $m$  | $r_i$ | $z$      | $|L(X_i)|$ |
|----------|-----|------|-------|----------|----------|
| AL-4K    | 37  | 4,000| 2/4   | 2,654    | 1/281    |
| AL-10K   | 37  | 10,000| 2/4  | 5,636    | 1/648    |
| HF-10K   | 56  | 10,000| 2/11 | 3,941    | 4/353    |
| USCD     | 56  | 10,000| 2/18 | 44,804   | 3/3857   |
| HEPAR II | 70  | 4,000| 2/4   | 1,714    | 1/381    |

ing Repository [95], including Alarm (AL), Hail Finder (HF), the US Census Data (UCSD) and HEPAR II. These are commonly considered too challenging to be solved exactly using sequential techniques, and are among the most demanding tests for the parent set assignment. The properties of all data sets are provided in Table 5.1, including: $r_i$ – the number of states (arity) that the variables in the set can assume, $z$ – the total number of generated maximal parent sets, and the properties of the output collection of the maximal parent sets $L(X_i)$.

5.4.1 Scalability Tests

In the first set of experiments, we analyzed scalability of the platform depending on the number of input variables $n$ and the number of observations $m$. We executed our Spark software on the varying number of nodes and we recorded wall time, as well as: $l_{max}$ – the deepest processed layer in the dynamic programming lattice, $l_z$ – the last layer at which we found new maximal parent sets, and the amount of extra work we had to perform due to removed synchronization (Section 5.3.2.2). The results of this experiment are summarized in Tables 5.2-5.3 and Figure 5.2. Here we report only relative speedup computed with respect to the time obtained on two nodes, since, except of AL-4K, we were not able to process the test data sets using sequential software.
Table 5.2: Execution time in minutes.

<table>
<thead>
<tr>
<th>Name</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL-4K</td>
<td>27.1</td>
<td>14.3</td>
<td>7</td>
<td>4.8</td>
<td>4.8</td>
<td>3.1</td>
<td>2.9</td>
</tr>
<tr>
<td>AL-10K</td>
<td>241.4</td>
<td>123.4</td>
<td>61</td>
<td>41.5</td>
<td>31.3</td>
<td>25.11</td>
<td>20.9</td>
</tr>
<tr>
<td>HF-10K</td>
<td>425.4</td>
<td>214</td>
<td>107.5</td>
<td>71.8</td>
<td>54.2</td>
<td>43.4</td>
<td>36.6</td>
</tr>
<tr>
<td>USCD</td>
<td>995.7</td>
<td>496.5</td>
<td>249.2</td>
<td>167.6</td>
<td>127.1</td>
<td>101.8</td>
<td>85.4</td>
</tr>
</tbody>
</table>

Table 5.3: Effect of reorganizing the search space.

<table>
<thead>
<tr>
<th>Name</th>
<th>$l_{\text{max}}$</th>
<th>$l_z$</th>
<th>Nodes Processed</th>
<th>Extra work</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL-4K</td>
<td>9</td>
<td>4</td>
<td>$4.6 \times 10^7$</td>
<td>1.68%</td>
</tr>
<tr>
<td>AL-10K</td>
<td>10</td>
<td>5</td>
<td>$1.6 \times 10^8$</td>
<td>1.72%</td>
</tr>
<tr>
<td>HF-10K</td>
<td>7</td>
<td>4</td>
<td>$3.1 \times 10^7$</td>
<td>3.2%</td>
</tr>
<tr>
<td>USCD</td>
<td>8</td>
<td>6</td>
<td>$2.1 \times 10^8$</td>
<td>1.23%</td>
</tr>
</tbody>
</table>

We start the analysis by first looking at the execution time and the speedup of our method. From Table 5.2 and Figure 5.2 we can see that, with the exception of AL-4K, the software achieves very good scalability on up to 24 nodes (480 cores). In all test cases, the required main memory never exceeds 84 GB, which enables us to run completely in the BFS mode. The slightly weaker scalability for AL-4K can be attributed to the overall size of the problem. With small $n$ and relatively small number of observations the problem can be solved in a few minutes on 12 nodes. Beyond that point, the overhead of synchronization between layers becomes significant compared to the optimized compute time on the collapsed dynamic programming lattice. As the number of observations for this data set increases (data set AL-10K), computational time increases and expectedly scalability improves.

Because of the relatively small size, we were able to process AL-4K using a sequential code in 2,435 minutes. The sequential code is written entirely in
C++, is optimized for memory usage and provides the same lattice constraining techniques as the parallel version. It also uses the same computational kernel to compute $s$. While this result suggests outstanding super-linear performance of our parallel code, we should keep in mind that the comparison is not completely fair, since the sequential version has significant overheads due to memory management (to avoid enumerating unnecessary tasks). Nevertheless, the result shows that even “small” problems can take more than a day to process sequentially, and this time can be easily reduced to minutes by using a cluster with a few nodes.

Next we consider the effectiveness of our approach in terms of removing synchronization and constraining the search space. Table 5.3 shows that in the worst case we have to perform only 3.2% of extra work compared to the completely synchronized version in which no unnecessary tasks are generated. At the same time, the total number of processed nodes is a small fraction of what we would have to process without constraining the search space. For exam-
ple, for the USCD benchmark the total number of tasks up to layer $l_{max} = 8$ is 1,689,096,333 when no pruning is applied, and it is reduced to approximately 12% of that when the pruning is enabled. Even then however, the total number of tasks to process is of the order of $10^8$, which demonstrates complexity of the problem and the need for parallel approach. The same pattern holds true for other tested data sets.

The last layer at which we enumerate new maximal parent sets, $l_z$, is always much smaller than $l_{max}$. This suggests that there is a room to tighten the pruning conditions. At the same time, it confirms that switching to the DFS at the higher levels of the dynamic programming lattice will not trigger any significant work due to how we manage the global state with all maximal parent sets. Finally, by looking at the results for AL-4 and AL-10 data sets, we can see that both $l_{max}$ and $l_z$ are increasing when the number of observations increases. This is because with the growing $m$ the network complexity term increases logarithmically for any variable $X_i$, but $s(X_i, \emptyset)$ grows linearly. Consequently, the effectiveness of the pruning conditions decreases. Nevertheless, the overall performance of the method remains reasonable.

### 5.4.2 Hepar II Test

In our second experiment, we focused on the HEPAR II test data. This benchmark comes from one of the early clinical decision support systems for multiple-disorder diagnosis of liver that involved a complex BN [91]. As we already mentioned, the parent set assignment problem plays a critical role in the exact BNSL, and hence directly translates into our ability to build high quality models for critical applications. This makes the benchmark interesting from the practi-
cal point of view. The benchmark is also challenging as it contains 70 variables, and all variables assume only few states, making it hard to identify variables that should be pruned.

To process HEPAR II we used all 25 nodes of our cluster. The experiment took 20 hours and 17 minutes to complete, with $l_{max} = 8$ and $l_z = 4$. To the best of our knowledge, this is the first time exact results for HEPAR II are reported. The peak memory consumption was 327 GB, and the execution involved the total of 10,770,519,474 tasks. Because the total memory available in our cluster is 1.6 TB, we again were able to process this benchmark completely in the BFS mode. However, to see how turning into the DFS mode affects the performance, we limited the available memory to 4 GB per node or 100 GB total memory. In this case, at layer $l = 7$ we had to switch to the DFS mode to process the remaining 9,427,586,763 tasks. This had the minor impact on the performance, and we were able to complete the entire execution in 20 hours and 28 minutes. Here we should keep in mind that because $l_z = 4$ there were no new maximal parent sets discovered when running in the DFS mode. However, we believe that even with new parent sets discovered the performance would not be drastically changed.

5.5 Related Work

Because of its importance, the parent assignment problem has been considered as a standalone question [16, 29] and in the relation to the structure learning of BNs [54, 58]. In [16], Koivisto provides several hardness results that suggest that the parent assignment for a single variable most likely has no polynomial-time solution. This motivates our parallel approach as there is a practical need to push the size of the problems that can be solved exactly in realistic time limits.
In [54, 58], multiple authors discuss the application of maximal parent sets in exact BNs structure learning. However, in each case maximal parent sets are assumed to be given and no details of how that is achieved are provided. In this paper, we provide the actual scalable algorithm for maximal parent sets enumeration, which in fact can be combined with any BN structure learning strategy. There is a significant body of work on solving maximal parent sets enumeration while discovering BN structure [96, 97, 98], including parallel algorithms [60, 99]. However, when both problems are coupled many optimizations specific to the parent sets enumeration become infeasible. As a result, these combined approaches do not scale and are limited to the instances with 30 to 40 variables, even when using thousands of cores and provably optimal MPI-based realizations [60, 99]. Finally, recently Scanagatta et al. [29] proposed a greedy heuristic that depends on a fast approximation of the actual scoring function to constraint the number of explored parent sets. While this approach can be used to solve problems larger than what we report, it does not provide any quality guarantees. In contrast, our method is guaranteed to provide the exact solution.

5.6 Conclusions

The exact parent set identification is a challenging problem with important applications in the exact structure learning of BNs. In this paper, we proposed a new scalable distributed memory approach to the problem, and we used it to efficiently process HEPAR II data set. This experiment clearly demonstrated that our method can handle even the most challenging data sets and using only limited hardware resources. This in turn opens new possibilities for exact learning
of large BNs, as with some effort our method can be combined with the already existing solvers, e.g. [58]. Our approach is scalable and we believe can be generalized to other popular scoring functions including AIC and BDeu. Since the efficiency of constraining the search space for these functions is currently unclear, the ability of our solution to adopt to heavy workloads provides a significant advantage.
Optimal Path Extension

Overview. The problem of exact BNSL is known to be NP-hard and the existing methods are both computationally and memory intensive. Via this work, we introduce a new approach for exact structure learning that leverages relationship between a partial network structure and the remaining variables to constrain the number of ways in which the partial network can be optimally extended. Via experimental results, we show that the method provides up to three times improvement in runtime, and orders of magnitude reduction in memory consumption over the current best algorithms.

6.1 Introduction

While BNs offer multiple advantages in how they represent probabilities and how they explicitly handle uncertainty, they also pose challenges. This is because both structure learning and probabilistic inference in BNs have been demonstrated to be NP-complete [15, 100]. The computational complexity of discovering BN structure becomes prohibitive for large data and exact learning algo-
rithms have to be replaced by approximations or heuristics [1] which may not acceptable in some scenarios due to reasons explained in Chapter 3. These algorithms do not provide guarantees on the quality of the structures they find. At the same time, in many real-life scenarios finding the optimal BN structure is a necessity, for example to make different models (e.g. BNs learned under different statistical criteria) comparable, or to allow for precise reasoning about models’ performance.

To address this challenge, we introduce a new method to accelerate a scoring-based exact structure learning of BNs. Our method, which we call optimal path extension, leverages shortest-path formulation of the BN structure learning. It takes advantage of the relationship between a partial network structure and the remaining variables to constraint the number of ways in which the partial network can be optimally extended. This has the effect of “compacting” the dynamic programming lattice explored during the structure search, thus practically reducing computational and memory complexity. The technique is general and can be combined with various BN search strategies, such as BFS or different variants of the A-star algorithm. Through experimental results, we show that the method provides up to three times improvement in runtime, and orders of magnitude reduction in the memory consumption over the current best algorithms. Thus, our approach significantly expands the range of applications in which the exact BN structure learning can be applied, including for big data analytics.

In this dissertation work, we are considering the exact score-based structure learning problem, which is to find an optimal\(^1\) structure \(G\) given a scoring func-

\(^1\)We consistently use “an optimal” and not “the optimal” as multiple optimal solutions may exist.
tion $\text{Score}(G:D)$.

The remainder of this chapter is organized as follows: in Section 6.2 we introduce our proposed method, and we demonstrate its experimental validation in Section 6.3. We conclude the chapter in Section 6.4.

## 6.2 Proposed Method

The starting point for solving our optimization problem are recurrences in Equations 2.1 and 6.1. The standard dynamic programming approach involves memoization of both $Q^*$ and $d$. If we imagine dynamic programming as progressing in the top-down manner over the lattice $L$, then the memory complexity of memoization is $\Theta \left( \binom{n}{\frac{n}{2}} \right)$, which is the number of nodes in the largest layer of $L$. This quickly becomes prohibitive for any but small number of variables. On the computational side, dynamic programming requires that all edges in the lattice are visited, which implies $\Theta \left( n \cdot 2^n \right)$ steps. By casting the problem into the shortest path formulation we gain the flexibility of considering both recurrences independently, such that the memory and time complexity are reduced.

To address the complexity of computing and storing $d$, we use the concept of parent graph [54]. To construct parent graph, we use the parent set assignment solver as a sub-routine which enumerates all maximal parent sets. For each $X_i$ we maintain an ordered vector of tuples $(U, s(X_i, U))$ (see example in Figure 6.1b) obtained from the invoked parent set assignment solver. Tuples are sorted in the ascending order of $s$, and we use binary encoding to represent $U$. The binary representation allows for $O(1)$ set containment and set equality checking as long as the number of variables does not exceed the word size of the executing hardware (e.g. $n \leq 64$ on a 64-bit architecture). By keeping vec-
tors ordered, we can get the optimal choice of parents, and the corresponding score, for \( X_i \) in \( O(1) \), and we can answer arbitrary query \( d(X_i, U) \) in \( O(l) \), where \( l \) is the size of the vector for \( X_i \). This is because for a given \( X_i \) its optimal parents set will be stored as the first entry of the corresponding sorted vector, and to answer \( d(X_i, U) \) we have to find the first maximal parents set that is a subset of \( U \). Thereby, after constructing parent graph we can answer efficiently all queries \( d(X_i, U'') \) for any \( U'' \supseteq U \). If \( U'' \) is one of the maximal parents sets of \( X_i \) we simply return stored \( s(X_i, U'') \). Otherwise, \( d(X_i, U'') \) must be equal to the smallest \( s \) among all maximal parents sets for which \( U'' \) is a superset. Because the number of all maximal parents sets is much smaller than the entire dynamic programming lattice. Depending on the scoring function and the input data the reduction might be by orders of magnitude (see for example Table 6.1 in Section 6.3).

### 6.2.1 Optimization Problem

For the given set of random variables \( \mathcal{X} \) and a scoring function \( \text{Score}(G: D) \) that we want to minimize. The search space of all potential network structures is super-exponential and consists of

\[
C(n) = \sum_{i=1}^{n} (-1)^{i+1} \binom{n}{i} 2^{i(n-i)} C(n - i)
\]

DAGs with \( n \) nodes.

Because relative ordering of parents of \( X_i \) is irrelevant and the scoring function is decomposable, we can leverage dynamic programming to constraint the search space. This general idea has been exploited in different variants, for example in \([96, 60, 97, 98]\), and works as follows. Because any DAG must have at
least one sink node (i.e. a node without descendants), we can first identify an optimal sink and find its optimal parents assignment (i.e. its optimal parents set). Then, we can continue with the remaining nodes recursively organizing them into an optimal structure. Because we know that a sink node has no successors, it can be placed at the end of the topological order we are building.

Then, the optimal choice of a sink minimizes the sum of scores of sub-networks consisting of the sink and the remaining nodes. If we denote an optimal score of a network over \( U \subseteq \mathcal{X} \) by \( Q^*(U) \), then we have:

\[
Q^*(U) = \min_{X_i \in U} \left( d(X_i, U - \{X_i\}) + Q^*(U - \{X_i\}) \right),
\]

and by using dynamic programming to compute \( Q^*(\mathcal{X}) \) we can construct an optimal ordering \( \pi^*(\mathcal{X}) \).

The dynamic programming algorithm can be visualized as operating on the lattice \( L \) with \( n + 1 \) levels formed by the partial order “set inclusion” on the power set of \( \mathcal{X} \) [60, 97, 54] (see Figure 6.1a). Two nodes in the lattice, \( U' \) and \( U \), are connected only if \( U' \subset U \) and \( |U| = |U'| + 1 \). Here we use \( U \) to denote both a subset of \( \mathcal{X} \) and the corresponding node in the lattice \( L \). An important property of the lattice is that any path from its root to one of its nodes is equivalent to a specific ordering of variables in that node. Moreover, an edge \((U', U)\) corresponds to evaluating \( d(U - U', U') \). For instance, the path marked in Figure 6.1a represents ordering \( \pi(\mathcal{X}) = [X_3, X_2, X_4, X_1] \), and edge \((\{X_3\}, \{X_2, X_3\})\) means computing \( d(X_2, \{X_3\}) \). In [54] Yuan et al. observed that finding an optimal ordering (i.e. an optimal network structure) is equivalent to finding a shortest path from the root to the sink in the dynamic programming lattice (which they call an order graph). Because this formulation gives a significant flexibility in the
Figure 6.1: (a) Dynamic programming lattice for the problem with four variables, (b) example parent graph structure where for each variable $X_i$ an ordered vector of tuples $(U, s(X_i, U))$ is stored, and (c) the constrained lattice created via our optimal path extension technique as prescribed by the parent graph in (b). Let path marked in bold be the optimal solution. By constraining the dynamic programming lattice, after discovering node $\{X_3\}$ a search algorithm can follow directly to the final node.

6.2.2 Optimal Path Extension

With the parent graph available we can focus now on the second recursion, Equation 6.1, to find an optimal network score and hence optimal network structure. As we already explained, the problem is equivalent to finding a shortest path from the root to the sink of the corresponding dynamic programming lattice $L$. As previously, the challenge is due to the immense size of $L$. 
Let $Q(U, \pi)$, defined as:

$$
Q(U, \pi) = \sum_{X_i \in U} d(X_i, \{X_j | X_j \prec X_i \text{ in } \pi(U)\}),
$$

be the score of a network over set $U \subseteq \mathcal{X}$ prescribed by the topological ordering $\pi(U)$. From Equation 2.2, equivalently, $Q(U, \pi)$ is the length of the path from the root of the lattice $L$ to the node $U$ that yields ordering $\pi(U)$ (recall that we use $U$ to denote both a subset of $\mathcal{X}$ and a node in $L$). To find the desired shortest path in $L$, and hence $Q^*$ and $\pi^*$, we could use any shortest path solver ranging from BFS to A-star and its variants (e.g. Iterative Deepening Search). However, in all cases memory constraint becomes a limiting factor. For example, in BFS at least two consecutive layers of $L$ have to be maintained in memory, and in A-star open and closed lists may grow excessively depending on the quality of the heuristic function used. Consequently, to scale up it is critical to further constraint the search space, i.e. reduce the number of nodes that have to be considered in the dynamic programming lattice. To achieve this, we introduce the optimal path extension technique.

Consider a node $U$ at the level $k$ in the lattice $L$ (the root of the lattice is at level $k = 0$). This node has $k$ incoming edges and $n - k$ outgoing edges. Each of the outgoing edges corresponds to one particular way in which $U$, and thus any of its corresponding orderings/paths can be extended. However, in many cases we can immediately identify the only extensions that can lead to the optimal path from $U$ to the sink of the lattice. Since other extensions of $U$ will be suboptimal, we can safely remove them from consideration as they cannot be a part of the final shortest path. To identify a node that can be optimally extended we use the following observation. If $U$ is a superset of the optimal
parents set of $X_i$, then by definition of $d$ no variable can be added to $U$ such that the score $d(X_i, U)$ is improved. Moreover, to maintain topological ordering, $X_i$ must be preceded by all variables in $U$. Consequently, any optimal path from $U$ to the sink of the lattice must include edge from $U$ to $U \cup \{X_i\}$. This intuition is captured in the following theorem:

**Theorem 1** (Optimal Path Extension). Let $U$ be a superset of the optimal parents set for $X_i \in \mathcal{X} - \{X_i\}$. Then, in the optimal path from $U$ to the sink of the dynamic programming lattice $U$ must be followed by $U \cup \{X_i\}$.

**Proof.** Let $\pi_1(\mathcal{X}) = \pi(U) \sim X_i \sim X_j \sim \pi(V)$ and $\pi_2(\mathcal{X}) = \pi(U) \sim X_j \sim X_i \sim \pi(V)$, where $V = \mathcal{X} - U - \{X_i, X_j\}$, represent two possible paths from the source to the sink of the dynamic programming lattice. We have that

$$Q(\mathcal{X}, \pi_1) = Q(U, \pi) + d(X_i, U) + d(X_j, U \cup \{X_i\}) + R$$

and

$$Q(\mathcal{X}, \pi_2) = Q(U, \pi) + d(X_j, U) + d(X_i, U \cup \{X_j\}) + R,$$

where $R$ is the length of the shortest path from $U \cup \{X_i, X_j\}$ to the sink of the lattice. Because $U$ is the superset of the optimal parents set for $X_i$ we have $d(X_i, U \cup \{X_j\}) = d(X_i, U)$. Now we consider two cases. If $U$ is not optimal parents set for $X_j$, we have $d(X_j, U) \geq d(X_j, U \cup \{X_i\})$ and it follows that:

$$Q(\mathcal{X}, \pi_2) \geq Q(U, \pi) + d(X_j, U \cup \{X_i\}) + d(X_i, U) + R$$

$$\geq Q(\mathcal{X}, \pi_1),$$

and hence $\pi_2$ is not optimal. If $U$ is the optimal parents set for $X_j$, then both
paths become equivalent and optimal.

To better illustrate the optimal path extension idea, consider example dynamic programming lattice and parent graph presented in Figure 6.1b. The optimal parent set of $X_2$ and $X_4$ consists of $X_3$ only. Now take node $U = \{X_3\}$. Since $U$ is a superset of the optimal parent set of $X_2$ and $X_4$, from Theorem 1 to extend $U$ it is sufficient to consider one of only these two variables. Suppose that we extend $U$ by adding $X_2$. The new node $\{X_2, X_3\}$ with the ordering $[X_3, X_2]$ remains the superset of the optimal parent set of $X_4$. Thus, we can further extend $\{X_2, X_3\}$ by adding $X_4$ with the corresponding ordering $[X_3, X_2, X_4]$. In the final step, we can extend one more time by adding $X_1$, hence reaching the sink of the lattice. In a similar way, we can extend $\{X_1, X_4\}$ by including $X_3$ and then $X_2$. In some cases extension will not be possible. For example, nodes $\{X_1\}$ and $\{X_1, X_2\}$ cannot be extended as no variable has optimal parents set that would be a subset of either of them. If we consider all possible path extensions, then the final compacted lattice will be reduced by one node and seven edges (see Figure 6.1c).

By applying our path extension technique we can significantly reduce the number of nodes and edges that have to be considered in the dynamic programming lattice. The extent to which reduction can be performed depends on the size of the optimal parents set of each variable – smaller the optimal parents set, higher the chance that the optimal path extension can be applied. Moreover, the effectiveness of our method will be higher for larger problems (i.e. problems with larger $\mathcal{X}$) because the dynamic programming lattice will include more nodes with a potential to extend. While at this moment we do not have complete theoretical bound on the expected number of nodes and edges that
can be removed via the path extension technique, our experimental results in Section 6.3 show excellent performance in practice.

6.2.3 Searching via Optimal Path Extension

The net effect of using our optimal path extension technique is compaction of the dynamic programming lattice. However, it would be counterproductive to first build the lattice and then apply the technique. Instead, the optimal path extension can be efficiently combined with any shortest path solver. To show how, we use the classic A-star search with a simple heuristic function. The function relaxes the BN acyclicity constraint and assumes that all variables not included in the currently explored node form a network by selecting optimal parents from among all other variables. Formally, the heuristic function is defined as 

\[ h(U) = \sum_{X_i \in \mathcal{X} \setminus U} d(X_i, \mathcal{X} - X_i). \]

The function is easy to implement and it is known to be consistent [54].

The resulting A-star procedure is outlined in Algorithm 4. To represent a search state we use simple structure with attribute \( g \) storing the exact distance from the root to the current node in the lattice and \( h \) storing the estimated distance from the current node to the sink of the lattice (as given by function \( h \)). For convenience, we store also \( f \), which is the sum of \( g \) and \( h \). The corresponding set of variables for a given state (i.e. the actual lattice node) is stored in attribute \( \text{set} \) using binary encoding as explained earlier. Finally, to reconstruct the optimal path and hence ordering we store also parent node information in attribute \( p \). We note that the final optimal network structure can be easily reconstructed from the parent graph and the shortest path information.

The algorithm follows the classic A-star pattern with \( Q \) representing open
list implemented as Fibonacci heap, and $C$ maintaining a closed list implemented as a simple hash table. The search states corresponding to the explored lattice nodes are generated on the fly in the loop in line 14. In lines 4, 15 and 16 we use the parent graph structure to extract values $d$. Recall that this requires a linear scan to obtain $d(X_i, v.set)$, and $O(1)$ to obtain $d(X_i, X - \{X_i\})$, which is the score of the optimal parents set for $X_i$. The cost of the linear scan in line 16 is in general negligible. This is because as the algorithm progresses the $v.set$ becomes larger and hence the probability of finding a relevant subset in the parent graph increases. The key element of the algorithm is the path extension procedure invoked in line 20. The procedure, outlined in Algorithm 5, returns a search state that can be reached directly from the current state via application of our path extension technique (so for example, for the node $\{X_3\}$ in Figure 6.1 it would return node $\{X_1, X_2, X_3, X_4\}$).

The path extension procedure iteratively applies Theorem 1 to the input node represented by $u.set$. First, it tests each variable $X_i$ to see whether the input node is a superset of the $X_i$’s optimal parents set (line 4). In practice, this requires one set containment check between $u.set$ and the set of variables stored in the first entry of the parent graph for $X_i$. If optimal path extension can be applied, state $u$ is updated and the process continues until no extension is possible. The final node is returned back to the main A-star procedure that follows without any changes.

From the computational complexity perspective, our approach includes a minimal overhead (e.g. a $O(n)$ linear scan in Algorithm 5) at the benefit of significantly constraining the number of nodes that have to be considered and stored in $Q$ and $C$. From the implementation perspective, only one small procedure has to be added to the A-star core. This holds true for other search algo-
Algorithm 4 A-star With Optimal Path Extension

1: \( s.g \leftarrow 0 \)
2: \( s.h \leftarrow 0 \)
3: for \( X_i \in X \) do
4: \( s.h \leftarrow s.h + d(X_i, X - \{X_i\}) \)
5: \( s.f \leftarrow s.h \)
6: \( s.set \leftarrow \phi \)
7: \( s.p \leftarrow \phi \)
8: \( Q.push(s) \)
9: while \( Q \neq \phi \) do
10: \( v \leftarrow Q.pop() \)
11: \( C.push(v) \)
12: if \( v.set = X \) then
13: \( \text{return Backtrack}(v, C) \)
14: for \( X_i \in X - v.set \) do
15: \( u.g \leftarrow v.g + d(X_i, v.set) \)
16: \( u.h \leftarrow v.h - d(X_i, X - \{X_i\}) \)
17: \( u.f \leftarrow u.g + u.h \)
18: \( u.set \leftarrow u.set \cup \{X_i\} \)
19: \( u.p \leftarrow v.set \)
20: \( u \leftarrow \text{PATHEXTENSION}(u) \)
21: if \( u \notin C \) then
22: if \( u \in Q \) then
23: \( pu \leftarrow Q.handle(u) \)
24: if \( pu.f > u.f \) then
25: \( pu \leftarrow u \)
26: \( Q.update(pu) \)
27: else
28: \( Q.push(u) \)

Algorithm 5 PATHEXTENSION

1: repeat
2: \( \text{extended} \leftarrow \text{false} \)
3: for \( X_i \in X - u.set \) do
4: if \( d(X_i, u.set) = d(X_i, X - \{X_i\}) \) then
5: \( u.g \leftarrow u.g + d(X_i, X - \{X_i\}) \)
6: \( u.h \leftarrow u.h - d(X_i, X - \{X_i\}) \)
7: \( u.set \leftarrow u.set \cup \{X_i\} \)
8: \( \text{extended} \leftarrow \text{true} \)
9: until \( \text{not extended} \)
10: \( \text{return } u \)
Algorithm 6 BFS With Optimal Path Extension

1: $s.g ← 0$
2: $s.set ← φ$
3: $s.p ← φ$
4: $N ← φ$
5: $N.insert(s)$
6: $C ← emptylist$
7: for $i ∈ \{1 . . . n\}$ do
8: $C.swap(N)$
9: $N.clear()$
10: for $v ∈ C$ do
11: for $X_i ∈ \mathcal{X} − v.set$ do
12: $u.g ← v.g + d(X_i, v.set)$
13: $u.set ← v.set ∪ \{X_i\}$
14: $u.p ← v.p$
15: $u.p.append(X_i)$
16: $u ← \text{PATHEXTENSION}(u)$
17: if $u ∉ N$ then
18: $N.insert(u)$
19: else
20: $pu ← N.handle(u)$
21: if $pu.f > u.f$ then
22: $pu ← u$
23: return $N.top()$

Algorithms as well. For example, in case of BFS the path extension procedure could be invoked for every node before that node is pushed into the FIFO queue. Finally, the method does not conflict but rather complements other possible optimizations such as exploring independencies between variables, which we do not discuss or consider in this work.

6.3 Experimental Results

We implemented our proposed method in the SABNA toolkit (Scalable Accelerated BN Analytics). Currently, the toolkit supports efficient parent graph con-
struction for all popular scoring functions like MDL, AIC, and BIC from any categorical data, and different optimal search strategies. It is written in C++14 and is available under the MIT License from the GitLab repository (https://gitlab.com/SCoRe-Group/SABNA-Release).

To understand the performance characteristics of our approach, we compared it with a top-down Breadth First Search (BFS) and the A-star search as implemented in the URLearning package version from 2016-05-17 [101]. We decided to use URLearning as this software provides some of the most advanced A-star search heuristics, and has been demonstrated to outperform other methods [54]. All tools were compiled using GCC 4.9.2 with standard optimization flags. To perform our tests we used a dedicated Linux compute server running in the exclusive mode under the Simple Linux Utility for Resource Management (SLURM). The server has dual 10-core Intel Xeon E5v3 2.3GHz processor and 64GB of RAM. However, in all tests only a single core was used to run the tested code with the remaining cores left to the operating system.

6.3.1 Test Data and Experimental Setup

To perform our tests we used a collection of standard benchmark datasets summarized in Table 6.1 [101]. To ensure that the results are comparable between SABNA and URLearning we used the following protocol. Because URLearning supports only binary variables, all datasets had been transformed into the \{0, 1\} domain, with 0 assigned to every value below the mean for a given variable and 1 if the value was above the mean. The parent graph for each dataset had been constructed using the URLearning tool with default parameters, and all methods ran with the same parent graph. In Table 6.1, we report the size of each parent
Table 6.1: Datasets used in the experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$n$</th>
<th>$m$</th>
<th>PG size</th>
<th>PG size reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>23</td>
<td>8,124</td>
<td>375,609</td>
<td>$2.6 \times 10^2$</td>
</tr>
<tr>
<td>Autos</td>
<td>26</td>
<td>159</td>
<td>2,391</td>
<td>$3.6 \times 10^5$</td>
</tr>
<tr>
<td>Insurance</td>
<td>27</td>
<td>1,000</td>
<td>1,518</td>
<td>$1.2 \times 10^6$</td>
</tr>
<tr>
<td>Water</td>
<td>32</td>
<td>1,000</td>
<td>328</td>
<td>$2.1 \times 10^8$</td>
</tr>
<tr>
<td>Soybean</td>
<td>36</td>
<td>266</td>
<td>5,926</td>
<td>$2.1 \times 10^8$</td>
</tr>
<tr>
<td>Alarm</td>
<td>37</td>
<td>1,000</td>
<td>672</td>
<td>$3.8 \times 10^9$</td>
</tr>
<tr>
<td>Bands</td>
<td>39</td>
<td>277</td>
<td>887</td>
<td>$1.2 \times 10^{10}$</td>
</tr>
</tbody>
</table>

Table 6.2: Runtime comparison of different methods.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BFS</th>
<th>URLearning</th>
<th>SABNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>2m12s</td>
<td>1m29s</td>
<td>1m</td>
</tr>
<tr>
<td>Autos</td>
<td>3m54s</td>
<td>37s</td>
<td>13s</td>
</tr>
<tr>
<td>Insurance</td>
<td>8m14s</td>
<td>7m25s</td>
<td>2m28s</td>
</tr>
<tr>
<td>Water</td>
<td>M</td>
<td>M</td>
<td>2m8s</td>
</tr>
<tr>
<td>Soybean</td>
<td>M</td>
<td>M</td>
<td>1h36m</td>
</tr>
<tr>
<td>Alarm</td>
<td>M</td>
<td>T</td>
<td>1h3m</td>
</tr>
<tr>
<td>Bands</td>
<td>M</td>
<td>T</td>
<td>1h10m</td>
</tr>
</tbody>
</table>

M – program ran out of memory. T – program ran out of time.

graph (PG) together with how its size is reduced compared to storing all values $d$. Finally, all tools were limited to the 64GB of RAM (i.e. no swap memory) and were terminated if they exceeded two hours runtime limit. In Tables 6.2–6.4 we summarize obtained results, with the runtime and memory usage averaged over 10 executions with a negligible variance. The runtime was measured via the system wall-clock, and approximate memory usage is based on the SLURM reports.
Table 6.3: Number of nodes visited by method ($\times 10^6$).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Lattice size</th>
<th>URLearning</th>
<th>SABNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>8.38</td>
<td>2.50</td>
<td>2.72</td>
</tr>
<tr>
<td>Autos</td>
<td>67.10</td>
<td>4.70</td>
<td>1.84</td>
</tr>
<tr>
<td>Insurance</td>
<td>134.21</td>
<td>57</td>
<td>13.52</td>
</tr>
<tr>
<td>Water</td>
<td>$4.29 \times 10^3$</td>
<td>–</td>
<td>13.03</td>
</tr>
<tr>
<td>Soybean</td>
<td>$6.87 \times 10^4$</td>
<td>–</td>
<td>330.50</td>
</tr>
<tr>
<td>Alarm</td>
<td>$1.37 \times 10^5$</td>
<td>–</td>
<td>217.37</td>
</tr>
<tr>
<td>Bands</td>
<td>$5.49 \times 10^5$</td>
<td>–</td>
<td>233.80</td>
</tr>
</tbody>
</table>

BFS has to visit all $2^n$ nodes in the lattice.

Table 6.4: Memory usage for different methods (in GB).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BFS</th>
<th>URLearning</th>
<th>SABNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>0.23</td>
<td>0.57</td>
<td>0.21</td>
</tr>
<tr>
<td>Autos</td>
<td>1.50</td>
<td>1.84</td>
<td>0.001</td>
</tr>
<tr>
<td>Insurance</td>
<td>2.99</td>
<td>10.67</td>
<td>1.07</td>
</tr>
<tr>
<td>Water</td>
<td>–</td>
<td>–</td>
<td>1.03</td>
</tr>
<tr>
<td>Soybean</td>
<td>–</td>
<td>–</td>
<td>27.16</td>
</tr>
<tr>
<td>Alarm</td>
<td>–</td>
<td>–</td>
<td>17.23</td>
</tr>
<tr>
<td>Bands</td>
<td>–</td>
<td>–</td>
<td>20.53</td>
</tr>
</tbody>
</table>

6.3.2 Results Discussion

We start the analysis by looking at the runtime of all three methods. Table 6.2 shows that SABNA is able to process all test datasets and it outperforms both BFS and the A-star strategy of URLearning, irrespective of the input dataset. As the number of variables increases, the performance difference becomes more pronounced, and for the largest dataset successfully processed by all methods (i.e. “Insurance”) SABNA is over three times faster than the other methods. Both SABNA and URLearning use the A-star search strategy. However, SABNA implements the most basic heuristic while URLearning employs a provably tighter heuristic with pattern database [54, 102]. In spite of this, SABNA explores signif-
icantly fewer states of the dynamic programming lattice, as shown in Table 6.3. This implies that the efficiency of SABNA should be attributed solely to our path extension technique. In fact, in additional tests not reported here, the basic A-star search performed only slightly better than BFS that visits all nodes in the lattice. This is significant considering how heuristic-sensitive is A-star. For example, because the URLearning heuristic is not well tuned to the data in “Water” and “Soybean” datasets, the open and closed lists of A-star explode and the method runs out of memory. By contrast, for the same datasets SABNA requires only 1GB and 27GB respectively (see Table 6.4). What is more, even for the largest datasets for which BFS and URLearning failed, SABNA consumed no more than 28GB of memory. This is well explained by the results in Table 6.3. With one exception, SABNA visits several times fewer nodes in the dynamic programming lattice than URLearning. This directly translates into a small memory footprint and clearly demonstrates the effectiveness of our approach. For the “Mushroom” dataset SABNA explores more nodes, yet it remains faster. This can be attributed to the overhead due to the pattern database construction in URLearning. At the same time, the overhead of our method is minimal. Because the “Mushroom” dataset is relatively small the overhead becomes the major component of the overall runtime. To summarize, presented experimental results consistently demonstrate that our path extension technique significantly reduces the number of states that have to be explored during the search process. This has the effect of reducing both memory and computational complexity such that much larger data can be processed.
6.4 Conclusions

In this chapter, we presented a new approach to accelerate the exact structure learning of BNs. Our experimental results demonstrated that the method performs extremely well in practice, even though it does not improve the worst case complexity. Our method is flexible and can be seamlessly combined with different search strategies. One of the main challenges in finding optimal BN structures is exponentially growing space complexity. While our method partially addresses this challenge, it can be further improved by expanding into distributed memory architectures (e.g. similar to our previous work [60]). This could open new range of applications for exact structure learning, including in clinical decision support systems or in genetics where problems with large number of variables are common.
Chapter 7

Conclusions and Open Problems

7.1 Conclusions

Learning high-quality large BNs is the key to developing causal BNs, and hence to building an AI system capable of human-level intelligence. The proposed methods advance that vision, by pushing the scale of exact BNs.

All of the proposed methods in the dissertation are made available as a part of the high-performance and well-documented software package SABNA (Scalable Accelerated Bayesian Network Analytics) that supports end-to-end BN analysis. The tool is accessible to both researchers and practitioners interested in using BNs in their projects. For example, currently SABNA is being used for medical data analysis, by the Department of Veteran Affairs (digital tumor board) and Jacobs School of Medicine and Biomedical Sciences to study opioids addiction.

Efficient implementation of the proposed scalable solutions in SABNA, enabled realization of exact structure learning for medium-large sized BN without resorting to distributed memory or expensive computing machines as depicted
in Figure 7.1. SABNA effectively pushes the boundary of what is feasible in exact BN structure learning.

![Figure 7.1: Problem space in exact Bayesian networks modeling. Currently, the entire domain of problem sizes is feasible using only our methods - SABNA.](image)

### 7.2 Open Problems

The proposed solutions not only extend the boundary in BN analytics, but also open several interesting research questions. Specifically:

- Better bounds on scoring functions in parent set assignment problem: Pruning of the search space in the parent sets assignment problem depends on tight bounds for the scoring functions. For example, experimental evidence suggests that there is significant room to improve bounds on MDL, by looking into pairwise relationships between variables.

- Inclusion of mixed variables and latent variables: In the current form, SABNA works under the assumption of complete categorical data and no latent variables. This is somewhat idealized scenario, as in the many practical applications the data may be incomplete. Incorporating support for (known) hidden variables poses exciting computational problems and hence is good target for modern parallel computers.
• Using exact algorithms to certify the quality of heuristics: Even though the proposed solutions are scalable, the theoretical computational complexity still catches up (see Figure 7.1). In such cases, heuristics are the only viable strategy. However, the is an open question of whether we could use small exact Bayesian networks as a tool to certify (or assess) the quality of a solution delivered by heuristics.

• Reinforcement structure learning with inference: Reinforcement learning provides the means to get causal insights about local interactions between variables. Combining reinforcement learning with exact structure learning could provide BNs with closer to causal networks.

• Use BNs to act as a meta-ML method for other methods: Other ML models, especially popular deep-learning methods, which are excellent at identifying association but terrible at expressing causation, could be combined with BN, with the goal of taking best of two worlds. BNs can provide insights or explanation into why deep model performed this and not that way, while deep models could potentially guide construction of BNs.


