Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

1-1-2012

Learning Optimal Bayesian Networks with Heuristic Search

Brandon M. Malone

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
Malone, Brandon M., "Learning Optimal Bayesian Networks with Heuristic Search" (2012). *Theses and Dissertations*. 2937.
https://scholarsjunction.msstate.edu/td/2937

This Dissertation - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

Learning optimal Bayesian networks with heuristic search

By

Brandon M. Malone

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulf llment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2012

Copyright by

Brandon M. Malone

2012

Learning optimal Bayesian networks with heuristic search

By

Brandon M. Malone

Approved:

Changhe Yuan
Assistant Professor of Computer
Science and Engineering
(Major Professor)

Susan Bridges
Professor Emeritus of Computer
Science and Engineering
(Committee Member)

Andy Perkins
Assistant Professor of Computer
Science and Engineering
(Committee Member)

Zhaohua Peng
Associate Professor of Biochemistry
and Molecular Biology
(Committee Member)

Edward B. Allen
Associate Professor of Computer
Science and Engineering,
and Graduate Coordinator

Sarah A. Rajala
Dean of the James Worth Bagley College
of Engineering

Name: Brandon M. Malone

Date of Degree: August 11, 2012

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Changhe Yuan

Title of Study: Learning optimal Bayesian networks with heuristic search

Pages of Study: 135

Candidate for Degree of Doctor of Philosophy

Bayesian networks are a widely used graphical model which formalize reasoning under uncertainty. Unfortunately, construction of a Bayesian network by an expert is time-consuming, and, in some cases, all experts may not agree on the best structure for a problem domain. Additionally, for some complex systems such as those present in molecular biology, experts with an understanding of the entire domain and how individual components interact may not exist. In these cases, we must learn the network structure from available data. This dissertation focuses on score-based structure learning. In this context, a scoring function is used to measure the goodness of fit of a structure to data. The goal is to find the structure which optimizes the scoring function.

The first contribution of this dissertation is a shortest-path finding perspective for the problem of learning optimal Bayesian network structures. This perspective builds on earlier dynamic programming strategies, but, as we show, offers much more flexibility.

Second, we develop a set of data structures to improve the efficiency of many of the integral calculations for structure learning. Most of these data structures benefit our algorithms, dynamic programming and other formulations of the structure learning problem.

Next, we introduce a suite of algorithms that leverage the new data structures and shortest-path finding perspective for structure learning. These algorithms take advantage of a number of new heuristic functions to ignore provably sub-optimal parts of the search space. They also exploit regularities in the search that previous approaches could not. All of the algorithms we present have their own advantages. Some minimize work in a provable sense; others use external memory such as hard disk to scale to datasets with more variables. Several of the algorithms quickly find solutions and improve them as long as they are given more resources.

Our algorithms improve the state of the art in structure learning by running faster, using less memory and incorporating other desirable characteristics, such as anytime behavior. We also pose unanswered questions to drive research into the future.

Key words: Bayesian networks, heuristic search

ACKNOWLEDGEMENTS

I thank Changhe Yuan, my advisor, for all of his help and encouragement in completing this dissertation. Without his advice and guidance, I would never have been introduced to the world of Bayesian networks and structure learning. When we began working together, I had very little understanding of the problem; however, he deftly balanced pointing me in the right direction and allowing me to explore on my own so that we ultimately arrived in right place. I am thankful for all of the late nights he spent writing, editing and revising papers and his help in putting together this dissertation.

I would also like to thank Susan Bridges who was my advisor when I first arrived at MSU. She really showed me a world of research that I enjoyed (and still enjoy) very much. She also spent many nights helping me finalize papers and presentations, and I owe a large part of my success here to her, as well.

The other members of my committee, Andy Perkins and Zhaohua Peng, always gave any help they could. They were also very understanding when I had to focus on my dissertation. Eric Hansen provided some very stimulating talks on the heuristic search aspect of my work, and I thank him for all of his time and help.

This work was supported in part by grant IIS-0953723 (CAREER grant) from the National Science Foundation and by EPS-0903787 (EPSCoR grant). The findings and opin-

ions in this dissertation belong solely to the author, and are not necessarily those of the sponsors.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1. INTRODUCTION 1

1.1 Representation 2
1.2 Notation and Terminology 4

2. BAYESIAN NETWORK ALGORITHMS 6

2.1 Inference 6
2.1.1 Factors 7
2.1.2 Prior Marginal Probability 8
2.1.3 Posterior Marginal Probability and Probability of Evidence 9
2.1.4 Most Probable Explanation (MPE) 9
2.1.5 Maximum A Posteriori Hypothesis (MAP) 10
2.1.6 Most Relevant Explanation (MRE) 11
2.2 Parameter Learning 12
2.2.1 Maximum Likelihood Estimate (MLE) 12
2.2.2 Expectation Maximization (EM) 13
2.3 Structure Learning 13
2.3.1 Constraint-based Learning Algorithms 14
2.3.2 Scoring Functions 16
2.3.2.1 Minimum Description Length (MDL) 17
2.3.2.2 Akaike’s Information Criterion (AIC) 18
2.3.2.3 Bayesian Dirichlet with Score Equivalence and Uniform Priors (BDeu) 18
2.3.2.4 Factorized Normalized Maximum Likelihood (fNML) 19
2.3.3 Approximate Structure Learning Algorithms 20
2.3.3.1 Hill Climbing 21

2.3.3.2 Genetic Algorithms 21
2.3.3.3 Optimal Reinsertion 22
2.3.4 Optimal Structure Learning Algorithms 22
2.3.4.1 Restricted Structure 23
2.3.4.2 Mathematical Programming 23
2.3.4.3 Branch and Bound Search 24
2.3.4.4 Dynamic Programming 25
2.4 Other Problems 26
2.4.1 Constraints 26
2.4.2 Hidden Variables 26
2.4.3 Dynamic Bayesian Networks 27
2.4.4 Classif cation 27
2.4.5 Equivalence Classes 28

3. EVALUATION DESCRIPTION 29

3.1 Datasets 29
3.2 Other Implementations 29

4. HEURISTIC GRAPH SEARCH PERSPECTIVE 33

4.1 Learning Optimal Subnetworks 33
4.1.1 Forward Order Graph 34
4.1.2 Reverse Order Graph 35
4.2 Identifying Optimal Parent Sets 36
4.2.1 Full Parent Graphs 37
4.2.2 Sparse Parent Graphs 37
4.2.2.1 Construction 38
4.2.2.2 Eff cient Scanning 40
4.2.2.3 Memory Savings 41
4.3 Calculating Scores 44

5. BEST-FIRST HEURISTIC SEARCH 49

5.1 Heuristic Function 51
5.2 Successor Operator 52
5.3 Solution Reconstruction 53
5.4 Advantages of A* 53
5.5 Empirical Results 54

6. FRONTIER BREADTH-FIRST BRANCH AND BOUND SEARCH . . . 59

6.1 Branch and Bound 62

6.2 Coordinating the Graph Searches 63
6.3 Ordering the Scores on Disk 65
6.4 Duplicate Detection 67
6.5 Advantages of Frontier Breadth-First Branch and Bound 69
6.6 Empirical Results 70

7. ANYTIME DEPTH-FIRST BRANCH AND BOUND SEARCH 74

7.1 Anytime Algorithms 75
7.1.1 Weighted A* 75
7.1.2 Anytime Weighted A* 76
7.1.3 Anytime Repairing A* 76
7.1.4 Anytime Window A* 77
7.2 Anytime DFBnB Network Learning Algorithm 77
7.2.1 Incremental Sparse Updates 80
7.2.2 Closed List and Backups 81
7.2.3 Heuristic Function for the Reverse Order Graph 82
7.2.4 Repairing Inconsistent Nodes 83
7.2.5 Advantages of DFBnB 85
7.3 Empirical Results 85
7.3.1 Comparison of Anytime Behavior 85
7.3.2 Comparison of Running Time 87

8. THE K-CYCLE CONFLICT HEURISTIC 90

8.1 A Motivating Example 91
8.2 Dynamic k-cycle Conf ic Heuristic 93
8.3 Static k-cycle Conf ict Heuristic 96
8.4 Advantages of the k-cycle Conf ict Heuristic 98
8.5 Empirical Results 100
8.5.1 Improvement from the Pattern Database Heuristics 100
8.5.2 Results on Other Datasets 102

9. BOUNDED ERROR, ANYTIME, PARALLEL SEARCH 106

9.1 Parallel Best-First Search 107
9.1.1 Parallel Window Search 108
9.1.2 Parallel Retracting A* 108
9.1.3 Adaptive k-Parallel Best-First Search 109
9.1.4 Parallel Structured Duplicate Detection and Parallel Best-NBlock First Search 111
9.1.5 Parallel Frontier A* with Delayed Duplicate Detection . . 112
9.1.6 Parallel Dovetailing 113

9.2 BEAP Search Algorithm 114
9.3 Advantages of BEAP 115
9.4 Experimental Results 116
9.4.1 Node Expansions 117
9.4.2 Comparison of Anytime Behavior 119
9.4.3 Comparison of Solution Quality 120

10. CONCLUSIONS AND FUTURE WORK 122

10.1 Contributions 122
10.2 Future Work 123

REFERENCES 127

LIST OF TABLES

3.1 A description of all datasets used for evaluation in this dissertation. 31

3.2 A description of all external algorithm implementations used in this evaluation. 32

4.1 Sorted scores and parent sets for X_1 after pruning parent sets which are not possibly optimal. 41

4.2 The $parents_X(X_i)$ bit vectors for X_1. 42

4.3 The result of performing the bitwise operation to exclude all parent sets which include X_3. 42

4.4 The result of performing the bitwise operation to exclude all parent sets which include either X_3 or X_2. 42

4.5 Sparse parent graph algorithms. 43

4.6 Score calculation algorithm. 48

5.1 A* search algorithm. 50

6.1 A frontier BFBnB search algorithm. 60

7.1 A DFBnB search algorithm. 79

8.1 Dynamic k-cycle conf ict heuristic. 97

8.2 Static k-cycle conf ict heuristic. 99

8.3 A comparison of BFBnB and A* with various heuristics on $Auto$ and $Flag$. 103

8.4 A comparison of BFBnB and A* on several datasets using static pattern databases. 105

LIST OF FIGURES

1.1 A Bayesian network. 3

4.1 A forward order graph of four variables. 35

4.2 A reverse order graph of four variables. 36

4.3 A sample parent graph for variable X_1. 38

4.4 The maximum count of parent sets stored by each of the parent graph strategies. 45

4.5 An AD-tree. 47

5.1 Runtime comparison among BB, DP and A*. 56

5.2 Nodes expanded by A* at the middle layer of two datasets. 57

5.3 Comparison of the order graph nodes expanded by DP and A*. 58

6.1 Coordinating the parent and order graphs. 66

6.2 Examples of immediate and delayed duplicate detection. 68

6.3 Runtime comparison of DP, BFBnB and A*. 71

6.4 Comparison of order graph nodes stored in memory at once by DP and BFBnB. 72

6.5 Hard disk usage for the $WDBC$ dataset. 73

7.1 Anytime comparison of DFBnB, OR and BB. 88

7.2 Comparison on the runtimes of DP, BFBnB, DFBnB and BB to f nd optimal networks. 89

8.1 A directed graph representing the heuristic estimate for the start search node. 93

9.1 Order graph nodes expanded for each dataset and value of ϵ by BEAP. . . . 118

9.2 Convergence behavior or BEAP and BB. 119

9.3 The solution quality of networks learned by AWA*, BEAP and BB. 121

CHAPTER 1

INTRODUCTION

The proliferation of freely available repositories on the Internet has tremendously increased the amount of available datasets. For example, the Gene Expression Omnibus houses a wealth of data from biological experiments. Large-scale social networking information is available via the Facebook API. However, this information in not usable knowledge. Bayesian networks are a common machine learning technique used to represent general relationships from such datasets. When these relationships are not known *a priori*, the structure of the network must be learned. The goal of this dissertation is to improve the state of the art in learning Bayesian network structures by casting the problem as a heuristic graph search problem. We propose a variety of novel, eff cient data structures and algorithms to solve the learning problem.

The remainder of this chapter formally introduces Bayesian networks as well as notation and terminology used throughout the rest of this dissertation. Chapter 2 introduces several types of problems and algorithms for Bayesian networks, including those for inference, parameter learning and structure learning. We then present our heuristic graph search perspective for Bayesian network structure learning in which the start node maps to an empty Bayesian network, the goal node represents the optimal Bayesian network and intermediate search nodes correspond to optimal networks over subsets of variables.

The next 5 chapters describe novel heuristic graph search algorithms for learning optimal Bayesian network structures. In Chapter 4, we give an admissible heuristic function that optimistically estimates the distance from any intermediate node to the goal node. We then use that function to guide an A* search algorithm and ignore unpromising subnetworks. Next, we take advantage of regularity present within the learning problem to reduce the memory requirements compared to existing dynamic programming algorithms by expanding nodes in a breadth-f rst order. Furthermore, we use external memory to minimize the RAM requirements of the algorithm. We propose an anytime search algorithm in Chapter 6 that uses a different heuristic function. That algorithm in particular takes advantage of eff cient, sparse data structures to very quickly f nd good networks before ultimately f nding and proving the optimality of the best-scoring network. Chapter 7 focuses on improving the admissible heuristic by using pattern databases to calculate a tighter bound. The improved bound allows us to safely ignore more of the search space, which decreases both running time and memory requirements. The penultimate chapter is dedicated to parallel algorithms and discusses an anytime parallel algorithm with provable quality bounds. In comparison to other parallel structure learning algorithms, ours uses orders of magnitude less running time and memory. Finally, conclusions and future work close the dissertation.

1.1 Representation

A Bayesian network consists of a structural component specifying the relationships among concepts in a domain and a quantitative specif cation of those relationships [72]. The structural component of the network is a directed acyclic graph (DAG). Each of the

vertices corresponds to a random variable. A directed edge from a vertex X_i to another vertex X_j indicates a relationship between the two variables. X_i is called a parent of X_j. All of the parents of X_j are called $\mathbf{PA_j}$. The quantitative specif cation is a conditional probability distribution of each variable given its parents, $P(X_j|\mathbf{PA_j})$. Thus, the DAG represents a joint probability distribution factorized as $P(X_1 \ldots X_n) = \sum_{i=1}^{n} P(X_i|\mathbf{PA_i})$.

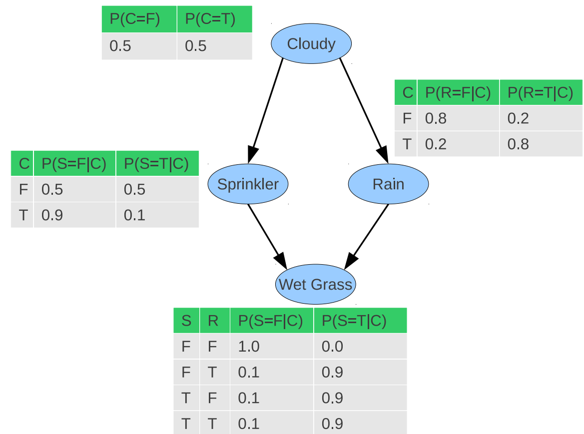


Figure 1.1

A Bayesian network.

Figure 1.1 displays an example Bayesian describing the domain of how weather can affect the grass. A topological sort of the vertices in the graph roughly corresponds to a causal interpretation of the domain. For example, clouds cause rain, which in turn causes the grass to be wet.

1.2 Notation and Terminology

The rest of this dissertation will use the following conventions. All variables are discrete. Uppercase letters (X) are random variables. Lowercase letters (x) are particular values, or instantiations, of those variables. Bold, uppercase letters ($\mathbf{V}$) are sets of random variables. Bold, lowercase letters ($\mathbf{v}$) are instantiations of those sets. Two instantiations are *consistent* if, for every variable the two have in common, those variables have the same values in both instantiations.

$P(X)$ and $P(\mathbf{X})$ are the probability distributions of a random variable and a set of random variables, respectively. $P(X|Y)$ is the conditional probability distribution of X given Y; either or both of X and Y could be sets of variables.

We denote the number of variables in a Bayesian network with n. Frequently, $\mathbf{V}$ will be used to refer to all of the variables in a Bayesian network. It is sometimes used in other contexts, though. The meaning should be clear from the context. Variable $X_i \in \mathbf{V}$ has r_i states.

A dataset $\mathcal{D}$ is a set of records $D_1 \ldots D_N$, where each record is an instantiation of the variables in $\mathbf{V}$. If all of the records instantiate all of the variables, then the dataset is *complete*; otherwise, it is *incomplete*. We use N to show the number of records in the dataset.

When discussing heuristic search, *node* refers to a node in the search graph. Bold, uppercase letters ($\mathbf{U}$) are also used to refer to nodes. When nodes in the search graph correspond to sets of variables, then the bold, uppercase letter is used to refer both to the node and the respective set of variables. The meaning should be clear from the context. When

referring specif cally to one of the vertices in the graphical Bayesian network structure, we say *vertex*.

CHAPTER 2

BAYESIAN NETWORK ALGORITHMS

Research in Bayesian networks can broadly be classif ed into several groups. Here, we consider inference, parameter learning and structure learning algorithms. Of course, other classif cations are possible. This chapter describes each of these three problems and algorithms for solving them.

2.1 Inference

Given a Bayesian network structure and parameters, inference is the problem of calculating the probability distribution of a subset of variables given values for some other (possibly overlapping) subset of variables [17]. For example, with the network from Figure 1.1, we may perform inference to answer the query ”What is the probability that it is cloudy given that the sprinkler is on?” This section will describe six types of inference queries: prior marginals, posterior marginals, probability of evidence, most probable explanation (MPE), maximum a posteriori hypothesis (MAP) and most relevant explanation (MRE). Computing both prior and posterior marginals are examples of belief updating. First, though, factors are described because they are a primitive structure in many of the inference algorithms.

2.1.1 Factors

A *factor* maps from an instantiation of a set of variables to a non-negative number [17]. Sometimes factors represent (conditional) probability distributions, but not always. The conditional probability tables of a Bayesian network always define a set of factors. Four main operations on factors are multiplying two factors; summing, or marginalizing, out a variable; maximizing out a variable; and reducing a factor given evidence.

Factor multiplication constructs a factor over the union of the variables in two other factors [17]. For example, suppose f_1 is a factor over X_1, X_2 and X_3, and f_2 is a factor over X_3 and X_4. Then the result of the multiplication $(f_1 f_2)$ is a factor over the variables X_1, X_2, X_3 and X_4. The values for the new factor are $(f_1 f_2)(\mathbf{z}) = f_1(\mathbf{x}) f_2(\mathbf{y})$ where $\mathbf{x}$ and $\mathbf{y}$ are consistent with $\mathbf{z}$. That is, the value in the new factor for a particular instantiation of X_1, X_2, X_3 and X_4 is equal to the value of f_1 for the instantiation of X_1, X_2 and X_3 times the value of f_2 for the instantiation of X_3 and X_4.

Marginalizing a variable from a factor removes one of the variables from that factor [17]. For example, suppose f_1 is a factor over X_1, X_2 and X_3. Marginalizing out X_3 results in a new factor over X_1 and X_2. The values of the new factor are $\left(\sum_X f\right)(\mathbf{y}) = \sum_x f(x, \mathbf{y})$. That is, the value in the new factor for a particular instantiation of X_1 and X_2 is equal to the sum of the values of f_1 consistent with the instantiation, regardless of the value of X_3.

Maximizing a variable from a factor is very similar to marginalizing it out of the factor [17]; the operation still removes one variable from the factor. If we again suppose f_1 is a factor over X_1, X_2 and X_3, then maximizing out X_3 results in a new factor over X_1 and

X_2. The values of the new factor are $(\max_X f)(\mathbf{y}) = \max_x f(x, \mathbf{y})$. That is, the value in the new factor for a particular instantiation of X_1 and X_2 is equal to the maximum of the values of f_1 consistent with the instantiation, regardless of the value of X_3. So the only difference between marginalizing and maximizing lies in taking either a sum or a max over the original factor.

Reducing a factor with evidence does not affect the variable domain of that factor [17]. So if f_1 is a factor over X_1, X_2 and X_3 and evidence $\mathbf{e}$ is an instantiation of X_3, then reducing f_1 given $\mathbf{e}$ results in the following.

$$f^{\mathbf{e}}(\mathbf{x}) = \left\{ \begin{array}{ll} f(\mathbf{x}), & \text{if} \mathbf{x} \sim \mathbf{e} \\ 0, & otherwise \end{array} \right\} \tag{2.1}$$

That is, the reduced factor returns the original value for instantiations consistent with the evidence and 0 otherwise.

2.1.2 Prior Marginal Probability

The prior marginal probability for a set of variables ref ects their probability distribution when no other information is given [17]. For example, for the network in Figure 1.1, we may ask ”What is the probability that the grass is wet?” Because the values of the Markov blanket variables (”Sprinkler” and ”Rain”) are unknown, we cannot directly extract this probability from the network representation.

To compute the prior marginal probabilities over a set of query variables $\mathbf{Q}$, we can marginalize all variables in $\mathbf{V} \setminus \mathbf{Q}$ using an algorithm called *variable elimination*. Given an elimination ordering π over the variables to marginalize and the factors corresponding

to the CPTs of the network, the algorithm iterates over each variable $\pi(i)$ in the ordering. All factors f_k which contain $\pi(i)$ are multiplied to f nd factor f. $\pi(i)$ is summed out of f, and all f_ks are replaced with $\sum_{\pi(i)} f$. After eliminating all variables in the ordering, the remaining factors are multiplied together to calculate $Pr(\mathbf{Q})$. The elimination order can dramatically affect the running time.

2.1.3 Posterior Marginal Probability and Probability of Evidence

Posterior marginal probabilities are similar to prior marginals, except that some evidence is given [17]. For Figure 1.1, we may ask ”What is the probability that the grass is wet given that it is cloudy?” Again, we cannot directly extract this probability from the network structure.

To compute posterior marginal probabilities over a set of query variables **Q** given evidence $\mathbf{e}$, we f rst calculate the joint marginal probabilities, $Pr(\mathbf{Q}, \mathbf{e})$. Joint marginals can be calculated using the variable elimination algorithm; however, rather than directly using the CPTs of the network as input, we instead reduce each of the CPT factors given $\mathbf{e}$. The resulting factor gives $Pr(\mathbf{Q}, \mathbf{e})$. By adding the numbers in the factor, we obtain $Pr(\mathbf{e})$. Consequently, normalizing the factor amounts to $\frac{Pr(\mathbf{Q},\mathbf{e})}{Pr(\mathbf{e})}$. This is the desired posterior marginal probability, $Pr(\mathbf{Q}|\mathbf{e})$.

2.1.4 Most Probable Explanation (MPE)

The MPE instantiation for some evidence **e** is the instantiation **q** of all variables not in **e** that maximizes the joint probability $P(\mathbf{e}, \mathbf{q})$ [17]. For Figure 1.1, we may ask “Which rain and sprinkler setting maximizes the probability that the grass is wet and it is cloudy?”

The Viterbi and Forward algorithms for Hidden Markov Models are applications of this type of inference.

To compute the MPE instantiation **q**, we use another slight adaptation of the variable elimination algorithm. We again reduce the CPT factors given **e**. Furthermore, rather than marginalizing out variables, we instead maximize them out. The resulting factor contains the probability of the MPE instantiation. We can easily extend factors to also track partial instantiations. Thus, the factor can also contain the actual MPE instantiation.

MPE can also be solved with heuristic search. Each search node corresponds to a partial instantiation. Successors of a node add one additional variable to the instantiation. The shortest path from the start node with no variables instantiated to a goal node with all variables instantiated corresponds to the MPE instantiation. Upper bounds can be calculated by introducing additional variables to the network.

Local search techniques have also been applied to identify MPE instantiations. In one common scheme, each state corresponds to a complete variable instantiation. Neighbors of a state change the instantiation of one variable. Hill climbing, for example, can be used to f nd a locally optimal instantiation given a start state.

2.1.5 Maximum A Posteriori Hypothesis (MAP)

The MAP instantiation for some evidence **e** is the instantiation **m** of some variables not in **e** that maximizes the joint probability $P(\mathbf{e}, \mathbf{m})$ [17]. For Figure 1.1, we may ask "Which sprinkler setting maximizes the probability that the grass is wet and it is cloudy?" MAP is

a generalization of MPE because the query is not restricted to all unobserved variables. In general, the MAP instantiation is not just a subset of the MPE instantiation.

Variable elimination can again be adapted to find the MAP instantiation. All CPT factors are reduced given **e**. Additionally, the elimination takes place in two phases. First, all non-MAP variables are marginalized out. Then, all MAP variables are maximized out. The resulting factor contains the MAP probability and corresponding instantiation for the MAP variables. The heuristic and local techniques described for MPE can also be adapted for MAP.

2.1.6 Most Relevant Explanation (MRE)

MPE always finds the most probable instantiations for all variables; MAP always find the most likely instantiation for a given set of variables. Other algorithms identify the instantiation of a single variable which best explains evidence. Often, though, we would like to pick the best explanation for evidence from among several different possible explanations which contain different variables. For example, in the fictitious Asia network [57], dyspnea could be caused either by visiting Asia and contracting tuberculosis or by having bronchitis. After identifying the best explanation, we do not care about the other variables. MPE always find the instantiations for all unobserved variables, while MAP can not selectively return the instantiation of one set of variables or the other. Single variable explanations, such as simply visiting Asia, cannot always fully explain evidence. MRE [101, 100, 99] is a framework which automatically identifies the best explanation according to a given relevance measure, such as genealized Bayes factor [32]. Like MAP,

MRE accepts as input evidence and a list of target variables. However, unlike MAP, MRE does not necessarily instantiate all of the target variables; rather, it finds an instantiation of the variables which maximizes the relevance measure. Therefore, in contrast to MPE and MAP, MRE does not instantiate target variables which are irrelevant to the best explanation. In contrast to single variable explanations, though, MRE can instantiate multiple variables if they best explain the evidence.

2.2 Parameter Learning

Given a network structure and a dataset, parameter learning is the problem of learning the conditional probability tables for each of the variables. In general, the probabilities are based on sufficient statistics (counts of particular instantiations) of the data. For example, for the network structure in Figure 1.1, if we were given a dataset instead of the probability tables, we may ask "What is the probability it will rain given that it is cloudy?" This section describes two methods for learning parameters: maximum likelihood estimate for complete datasets and Expectation Maximization for incomplete datasets.

2.2.1 Maximum Likelihood Estimate (MLE)

Suppose we are given a complete dataset $\mathcal{D}$ with N records. Assuming records are generated independently and according to their true distribution, then the *empirical distribution* $Pr_{\mathcal{D}}(\cdot)$ for instantiation **x** is the frequency of that instantiation within the dataset, $Pr_{\mathcal{D}}(\mathbf{x}) = \frac{\mathcal{D}\#(\mathbf{x})}{N}$, where $\mathcal{D}\#(\mathbf{x})$ is the number of records in $\mathcal{D}$ consistent with $\mathbf{x}$. $\mathcal{D}\#$ is also called a *sufficient statistic*. Suppose we have variable $X = x$ and its parents $\mathbf{U} = \mathbf{u}$. Then the MLE parameters [17] are estimated from the empirical distribution.

$$\theta^{ml}_{x|\mathbf{u}} = Pr_{\mathcal{D}}(x|\mathbf{u}) = \frac{\mathcal{D}\#(x, \mathbf{u})}{\mathcal{D}\#(x, \mathbf{u}} \tag{2.2}$$

The MLE parameters are the only estimates which maximize the likelihood function, $L(\theta|\mathcal{D}) = \prod_{i=1}^{N} Pr_{\theta}(\mathbf{d_i})$. For example, suppose we are given the structure in Figure 1.1 and a dataset in which $\mathcal{D}\#(Cloudy = true) = 10$ and $\mathcal{D}\#(Cloudy = true, Rainy = true) = 8$. Then the MLE parameter $Pr(Rainy = true|Cloudy = true) = 0.8$.

2.2.2 Expectation Maximization (EM)

The MLE parameters maximize the likelihood of the data; however, their calculation requires a complete dataset for the suff cient statistics. EM [25] is a technique for estimating suff cient statistics when datasets are missing values. EM starts with an initial (possibly random) set of parameters. It then alternates between an expectation phase and a maximization phase. In the expectation phase, the current parameters are used to estimate the missing values by performing inference in the current Bayesian network. This has the effect of completing the dataset, though some of the variable instantiations have fractional counts. The next iteration of MLE parameters are computed with the completed dataset. The new parameters are guaranteed to never have a smaller likelihood than the previous parameters. This process continues until the parameters converge.

2.3 Structure Learning

Two approaches have been proposed for learning the structure of Bayesian networks from data. One group of algorithms focuses on establishing conditional independence be-

tween variables using statistical tests such as the Chi-square test. A network structure is then constructed which maximizes the number of independencies discovered by the statistical tests.

The other group of algorithms focus on discovering a Bayesian network which optimizes a scoring function. The scoring function computes a measure of the goodness of f t of a network to a dataset [43]. The scoring functions all embody Occam's razor in one way or another.

In this section, we f rst describe constraint-based learning algorithms. Next, several commonly used scoring functions are described. We then present several algorithms which use approximate search methods to f nd networks. Finaly, we give a number of algorithms which guarantee to optimize a scoring function.

2.3.1 Constraint-based Learning Algorithms

Constraint-based algorithms begin with the observation that Bayesian networks encode conditional dependence relationships among the variables. Therefore, they f rst use a set of statistical tests, such as Chi-square or G-test, to establish which variables are conditionally independent from each other. These results of those tests are used to create a directed network structure. Examples of these algorithms include PC [86] and IC [93].

The PC algorithm begins with a complete, undirected graph over all of the variables. It then begins the indepence tests. First, it tests all pairs of variables for marginal indepence ($X \perp\!\!\!\perp Y$). Edges between marginally independent variables are removed. It then tests all pairs which still have edges between them for conditional independence by conditioning

on one variable ($X \perp\!\!\!\perp Y|A$). Edges between conditionally independent variables are removed. This process continues until the conditioning set of variables reaches a user-def ned size k. Finally, based on the result of the independence tests, some of the remaining edges are directed. Some edges may remain undirected because, for example, noise in the data could give contradictory test results.

The PC algorithm can be tractable. For each iteration of tests, each pair is tested at most once, so each iteration includes at most $O(n^2)$ tests. Furthermore, the largest possible conditioning set could be size $n - 1$, so there could be at most $O(n)$ iterations. Consequently, an upper bound on the number of required independence tests is $O(n^3)$. Other constraint-based algorithms reduce the number of independence tests compared to PC [23, 95]. Unfortunately, the algorithms are very sensitive to the results of the independence tests. The independence tests are in turn sensitive to the amount of available data. Often, though, we must learn in settings with limited data. Also, all of the independence tests require a user-specif ed signif cance threshold, which may not be easy to estimate *a priori*. Additionally, constraint-based algorithms do not have a Bayesian interpretation [43]. For these reasons, the rest of this dissertation does not consider constraint-based algorithms.

2.3.2 Scoring Functions

Many scoring functions are in the form of a penalized log-likelihood (LL) functions. The LL is the log probability of D given B. Under the standard i.i.d assumption, the likelihood of the data given a structure can be calculated as

$$LL(D|B) = \sum_{j}^{N} \log P(D_j|B) \tag{2.3}$$

$$= \sum_{i}^{n} \sum_{j}^{N} \log P\left(D_{ij}|PA_{ij}\right), \tag{2.4}$$

where D_{ij} is the instantiation of X_i in data point D_j, and PA_{ij} is the instantiation of X_i's parents in D_j. Adding an arc to a network never decreases the likelihood of the network. Intuitively, the extra arc is simply ignored if it does not add any more information. The extra arcs pose at least two problems, though. First, they may lead to overf tting of the training data and result in poor performance on testing data. Second, densely connected networks increase the running time when using the networks for downstream analysis, such as inference and prediction.

A penalized LL function aims to address the overf tting problem by adding a penalty term which penalizes complex networks. Therefore, even though the complex networks may have a very good LL score, a high penalty term may reduce the score to be below that of a less complex network. Here, we focus on decomposable penalized LL (DPLL) scores, which are always of the form

$$DPLL(B, D) = LL(D|B) - \sum_{i=1}^{n} Penalty(X_i, B, D). \tag{2.5}$$

The scores are all *decomposable* [43] because the score of the entire network is expressed as the sum of the scores of each variable. There are several well-known DPLL

scoring functions for learning Bayesian networks. We consider minimum description length (MDL) [80], Aikake's information criterion (AIC) [2], Bayesian Dirichlet with score equivalence and uniform priors (BDeu) [13, 43] and factorized normalized maximum likelihood (fNML) [83]. These scoring functions only differ in the penalty terms, so we will focus on the penalty terms in the following discussions.

A Bayesian network structure can represent a set of joint probability distributions. Two network structures are said to belong to the same equivalence class if they represent the same set of probability distributions [10]. A scoring function which assigns the same score to networks in the same equivalence class are *score equivalent* [43].

2.3.2.1 Minimum Description Length (MDL)

The MDL [80] scoring metric for Bayesian networks was def ned in [54, 87]. MDL approaches scoring Bayesian networks as an information theoretic task. The basic idea is to minimally encode D in two parts: the network structure and the unexplained data. The model can be encoded by storing the conditional probability tables of all variables. This requires $\frac{\log N}{2} * p$ bits, where $\frac{\log N}{2}$ is the expected space required to store one probability value and p is the number of individual probability values for all variables. The unexplained part of the data can be explained with $LL(D|B)$ bits. Therefore, we can write the MDL penalty term as

$$PenaltyMDL(X_i, B, D) = \frac{\log N * p_i}{2}, \tag{2.6}$$

where p_i is the number of parameters for X_i. For MDL, the penalty term ref ects that more complex models will require longer encodings. The penalty term for MDL is larger than

that of most other scoring functions, so optimal MDL networks tend to be sparser than those by other scoring functions. As hinted at by its name, an optimal MDL network minimizes rather than maximizes the scoring function. To interpret the penalty as a subtraction, the scores must be multiplied by -1. The Bayesian information criterion (BIC) [80] is a scoring function whose calculation is equivalent to MDL for Bayesian networks, but it is derived based on the asymptotic behavior of the models. That is, BIC is based on having a suff ciently large amount of data. Also, BIC does not require the -1 multiplication.

2.3.2.2 Akaike's Information Criterion (AIC)

Bozdogan [5] def ned the AIC [2] scoring metric for Bayesian networks. It, like BIC, is another scoring function based on the asymptotic behavior of models with suff ciently large datasets. In terms of the equation, the penalty for AIC differs from that of MDL by the $\log N$ term. So the AIC penalty term is

$$PenaltyAIC(X_i, B, D) = p_i \tag{2.7}$$

Because its penalty term is less than that of MDL, AIC tends to favor more complex networks than MDL.

2.3.2.3 Bayesian Dirichlet with Score Equivalence and Uniform Priors (BDeu)

The Bayesian Dirichlet (BD) scoring function was f rst proposed by Cooper and Herskovits [13]. It computes the joint probability of a network for a given dataset. However, the BD metric requires a user to specify a hyper parameter for all possible variable-parents combinations. Furthermore, it is not score equivalent, which requires assigning the same

score to equivalent structures. To address the problems, a single "hyperparameter" called the *equivalent sample size* is introduced, referred to as α [43]. All of the needed parameters can be calculated from α and a prior distribution over network structures. This score, called BDe, is score equivalent. Furthermore, if one assumes all network structures are equally likely, that is, the prior distribution over network structures is uniform, then α is the only input necessary for this scoring function. BDe with this additional uniformity assumption is called BDeu [43]. Somewhat independently, the BDeu scoring function was also proposed earlier by Buntine [6]. BDeu is also a decomposable penalized LL scoring function whose penalty term is

$$PenaltyBDeu(X_i, B, D) = \sum_{j}^{q_i} \sum_{k}^{r_i} \log \frac{P(D_{ijk}|D_{ij})}{P(D_{ijk}|D_{ij}, \alpha_{ij})}, \tag{2.8}$$

where q_i is the number of possible values of PA_i, r_i is the number of possible values for X_i, D_{ijk} is the number of times $X_i = k$ and $PA_i = j$ in D, and α_{ij} is a parameter calculated based on the user-specif ed α. The original derivations [6, 43] include a more detailed description. The density of the optimal network structure learned with BDeu is correlated with α; low α values typically result in sparser networks than higher α values. Recent studies [81] have shown the BDeu behavior is very sensitive to α. If the density of the network to be learned is unknown, selecting an appropriate α is diff cult.

2.3.2.4 Factorized Normalized Maximum Likelihood (fNML)

Silander *et al.* [83] developed the fNML score function to address the problem of α selection with BDeu. fNML is based on the normalized maximum likelihood function

(NML) [78]. NML is a penalized LL scoring function in which *regret* is the penalty term. Regret is calculated as

$$\sum_{D'} P(D'|B), \tag{2.9}$$

where the sum ranges over all possible datasets of size N. Kontkanen and Myllymäki [48] showed how to eff ciently calculate regret for a single variable. By calculating regret for each variable in the dataset, the NML becomes decomposable, or factorized. fNML is given by

$$PenaltyfNML(X_i, B, D) = \sum_{k}^{q_i} \log C_{N_{ij}}^{r_i}, \tag{2.10}$$

where $C_{N_{ij}}^{r_i}$ are the regrets. fNML is not score equivalent.

2.3.3 Approximate Structure Learning Algorithms

Learning a Bayesian network with a restricted number of parents for each variable which optimizes a particular scoring function is NP-complete [11]. Consequently, many early learning algorithms focused on approximate learning techniques which f nd local optima of the scoring function. Many approximate optimization techniques have been applied to learning Bayesian network structures. In general, these algorithms are based on a ”search-and-score” approach. A search algorithm, such as greedy hill climbing, identif es candidate structures. The structures are scored, and the best scoring structures are used to identify new candidates. The algorithm continues until converging to a locally optimal network structure. Hill climbing and genetic algorithms are two algorithms commonly applied to identify candidate structures. Optimal Reinsertion (OR) [64] is a more sophisticated hill climbing algorithms that performs well in practice [89, 91]

2.3.3.1 Hill Climbing

Hill climbing algorithms require three components: a scoring function, a start state and successor generation operators. Any of the Bayesian network score functions, such as BD or MDL, can serve as the scoring function for hill climbing. Each state in the search space corresponds to a single Bayesian network structure. The start state often corresponds to an empty network (with no edges); however, *a priori* knowledge can also be used to create a different starting network structure. Different algorithms use different successor generation operators. Three commonly used operators [53] are edge insertion, deletion and reversal. In basic greedy hill climbing, the highest scoring successor is retained and used to generate the next set of candidates. This methodology can easily be extended by keeping multiple highest scoring successors (turning it into a beam search) or avoiding previously generated candidates (tabu search).

2.3.3.2 Genetic Algorithms

Genetic algorithms have also been extensively used to identify candidate structures [56]. These algorithms typically require four main components: a fitness function, a chromosome representation, a crossover strategy and a strategy to generate the next generation. Furthermore, other parameters such as mutation rate and elitism also affect the performance of the algorithm. Any Bayesian network scoring function can serve as the fitness function. A chromosome in the genetic algorithm represents a complete Bayesian network structure. One possible representation is a bit string of length n^2. The first n bits indicate the parents of variable X_0; the next n bits indicate the parents for X_1, etc. After crossover,

chromosomes which do not correspond to valid Bayesian networks (because they have cycles, etc.) are removed from the population. The next generation of networks is then generated according to the generation strategy. Other evolutionary algorithms, such as ant colony optimization [16] and cooperative coevolution [3], have also been applied to this problem.

2.3.3.3 Optimal Reinsertion

The optimal reinsertion algorithm (OR) [64] is a hill climbing algorithm that uses a different operator: a variable is removed from the network, its optimal parents are selected, and the variable is then reinserted into the network with those parents. The parents are selected to ensure the new network is still a valid Bayesian network. While OR does select optimal parents locally, it does not guarantee to f nd the globally optimal structure. Often, a greedy hill climbing is run on the structure after OR reaches a local maximum to attempt to further improve its score.

2.3.4 Optimal Structure Learning Algorithms

The approximate search algorithms often run quickly; however, the learned network is of unknown quality. Thus, further interpretation of the learned structure must also account for this variance. This limitation led researchers to develop algorithms which learn networks which provably optimize a scoring function for a dataset.

2.3.4.1 Restricted Structure

The oldest optimal learning algorithm we consider [12] has existed for several decades; however, this algorithm only learns tree-structured networks in which variables have only a single parent. This algorithm reduces the structure learning problem to f nding the minimum spanning tree in a graph. A vertex in a fully connected graph represents each variable. The weight of each edge is equal to the mutual information between the two variables. The minimum spanning tree corresponds to an optimal tree network among the variables.

2.3.4.2 Mathematical Programming

Optimal networks have also been learned using mathematical programming (MP) [44, 15]. This technique reformulates the structure learning problem as a linear or integer program. An exponential number of constraints are used to def ne a convex hull in which each vertex corresponds to a DAG. The constraints are added incrementally as cutting planes. The algorithm alternates between two phases. In the f rst, it runs a linear or integer program and determines if the returned (optimal) solution is a vertex on the hull. If so, the algorithm terminates. Otherwise, it adds a number of cutting planes to exclude the found solution (as well as other non-DAG structures) from the feasible space of the program. Intuitively, the algorithm looks for clusers of nodes in the solution which are highly cyclic and adds cutting planes to exclude those cycles. The algorithm then returns to the f rst phase. Coordinate descent is used to identify the vertex which corresponds to the optimal DAG structure. Furthermore, the dual of their formulation provides an upper bound which can help guide the descent algorithm. This algorithm was shown to have similar runtime

performance as dynamic programming [44]. Implementations of MP have not been made available from the authors; therefore, none of our empirical comparisons include MP.

2.3.4.3 Branch and Bound Search

de Campos and Ji [19] proposed a systematic branch and bound search algorithm (BB) to identify optimal network structures. The algorithm begins by calculating optimal parent sets for all variables. These sets are represented as a directed graph that may have cycles. Cycles are then repeatedly broken by removing one edge at a time. The new (possibly cyclic) graphs correspond to nodes in a search space and are expanded in best-first order. Graphs that have been generated but not expanded are stored using a priority queue. Expanding a node consists of breaking a cycle in its graph. If a node does not have a cycle, its score is compared to the score of the best DAG so far. If its score is better, the network for that node becomes the new incumbent solution. Networks with lower bounds worse than the score of the current incumbent are not considered for expansion. The algorithm terminates when no more nodes need to be expanded. Their algorithm can also use simple constraints, such as "X can only have up to 3 parents" or "Y and Z must be parents of X."

They also add an anytime component to the algorithm by initially finding a solution using an approximation technique. Because they expand nodes in a best-first order, the most recently expanded node gives a lower bound on the globally optimal solution. Furthermore, in an attempt to find more acyclic graphs, their algorithm occasionally expands nodes with the worst score rather than the best. By considering the difference between the between the best network found so far and the lowest score of a node that has not yet

been expanded, they can bound the error of their solution. The algorithm also accounts for limited resources by switching to a depth-first search if the priority queue grows too large to fit in RAM. However, this algorithm was shown to be less effective than dynamic programming [21] for proving the optimality of networks.

2.3.4.4 Dynamic Programming

Dynamic programming algorithms find an optimal Bayesian network in $O(n2^n)$ time [84, 82]. The algorithms derive from the observation that, because the network is a DAG, the optimal structure contains a leaf variable (that has no children) and its parents, plus an optimal subnetwork over the other variables. This subnetwork is also a DAG. The algorithm recursively finds leaves of subnetworks to find the optimal complete network structure. Specifically, for a scoring function $Score(\cdot|\cdot)$ and variables $\mathbf{V}$ [84], the following equations give the recurrences.

$$Score(\mathbf{V}) = \min_{X \in \mathbf{V}} \{Score(\mathbf{V} \setminus \{X\}) + BestScore(X, \mathbf{V} \setminus \{X\})\} \tag{2.11}$$

$$BestScore(X, \mathbf{V} \setminus \{X\}) = \min_{PA_X \subseteq \mathbf{V} \setminus \{X\}} Score(X|PA_X). \tag{2.12}$$

$Score(\mathbf{V})$ gives the score for the subnetwork with variables $\mathbf{V}$. X is selected as the leaf of the subnetwork. $BestScore$ gives the best parents for X out of the remaining variables in the subnetwork $\mathbf{V}$. This recurrence suggests an algorithm starting with all variables and recursively removing one variable at a time. Each variable must be tried as the leaf of each subnetwork. Hence, $Score$ must be evaluated $O(2^n)$ times. Furthermore, each evaluation of $Score$ requires $O(n)$ calls to $BestScore$ to try each remaining variable as a leaf. This gives the $O(n2^n)$ time complexity. All of the intermediate results are stored in memory,

so the memory complexity is also $O(n2^n)$. Silander and Myllmaki [82] adapted the algorithm to instead begin with an empty subnetwork and recursively add leaves missing from $\mathbf{V}$. This does not change the recurrences, but was empirically shown [82] to be more computationally eff cient.

2.4 Other Problems

Not all problems surrounding Bayesian networks fall squarely into one of these categories. This section discusses some of these problems in more detail.

2.4.1 Constraints

The preceding discussion has assumed no prior information was available about the dataset. Often, human experts know some of the relationships between some of the variables. This prior knowledge can be used to constrain the learned network. Constraints are broadly characterizable along two axes. First, they can apply to either the structure [22] or the parameters [20]. Second, the constraints can either be hard [29] or soft [8]. Learned networks must respect hard constraints, while enough data can supersede soft constraints. Many algorithms have been proposed for incorporating constraints. This dissertation assumes no constraints on the learned network.

2.4.2 Hidden Variables

Bayesian networks represent a probability distribution over the variables $X_1 \ldots X_n$; however, it is possible that some other variables, not present in the dataset, also affect the probability distribution. For example, a hidden class variable could inf uence all of

the observed variables. In cases like these, if the network does not include the hidden variables, then it may not accurately model the probability distribution of the observed variables. If a hidden variable is known to exist, but simply unobserved, then it can be treated as a missing value. Relevant parameters can be estimated using EM [25]. In other cases, unknown hidden variables affect the probability distribution. Several algorithms exist for identifying hidden variables [34, 27, 26]. This work assumes no hidden variables exist which affect the probability distribution of the observed variables.

2.4.3 Dynamic Bayesian Networks

Because Bayesian network structures are restricted to DAGs, they are unable to capture cyclic relationships. This situation arises in, for example, learning gene regulatory networks [79]. In these networks, the protein products from gene g_1 can affect g_2 by being a transcription factor, for example. The protein products of g_2 can then affect g_1, creating a cycle. Dynamic Bayesian networks [39, 65] offer a solution to this problem. Dynamic Bayesian networks contain multiple vertices for each variable; each vertex corresponds to a different time slice. The gene regulatory relationships could be modeled in a dynamic Bayesian network with an edge from g_1 to g_2 in both time slices and an edge from g_2 in the f rst time slice to g_1 in the second time slice. This work does not consider these sorts of relationships.

2.4.4 Classif cation

Classif cation problems focus on learning a rule (classif er) which predicts the value of discrete variable (the class variables) given another set of variables (attributes). Regression

is a similar problem except the class variable is continuous. Typically, the only quality measure for a classif er is how accurately it predicts the class variable. In contrast, the discussed Bayesian network score functions, like MDL and BD, measure prediction accuracy across all variables (the likelihood of the data given the structure) as well as the complexity of the network. Consequently, a network optimizing a scoring function may not also optimize the prediction accuracy of the class variable. Several restricted structures, such as naive Bayes [55] and tree augmented naive Bayes [35], have been shown to outperform unrestricted network structures in prediction tasks. This work focuses on learning the relationships among all of the variables rather than predicting a single class variable.

2.4.5 Equivalence Classes

Many, though not all, scoring functions are score equivalent [10]. Score equivalent functions assign the same score to networks which represent the same probability distribution. The relationship among distributions with the same score is symmetric, transitive and ref exive; therefore, score equivalent functions partition the set of Bayesian networks into equivalence classes. Equivalent network structures [93] have the same skeleton (undirected graphical structure) and v-structures. Both the MDL and BDe score functions are score equivalent [10].

Optimal structure learning algorithms learn one member of the equivalence class with the optimal score. Chickering [10] describes an algorithm which identif es the equivalence class of a Bayesian network. This algorithm can extract the equivalence class of the structure found by the structure learning algorithm.

CHAPTER 3

EVALUATION DESCRIPTION

Throughout this dissertation, much of the evaluation procedure is the same. In general, we used data from the UCI machine learning repository [33], downloaded code from the authors for all comparisons, and ran on the same server. In order to prevent repeating the same material, we summarize all of this information here.

3.1 Datasets

Table 3.1 describes all of the UCI datasets used throughout this paper. To conserve space on the f gures in later chapters, we often abbreviate the dataset names. The table gives both the full name from the UCI repository as well as the abbreviation we use. It also lists the number of variables and number of records in the datasets.

3.2 Other Implementations

Table 3.2 describes all of the other implementations to which we compared. None of these algorithms are parallel learning algorithms. Both DP and BB include options to calculate local scores in parallel, but these options were never used. Unless otherwise noted, all other algorithms were implemented with custom code in Java.

All experiments were performed on a PC with a dual quad-core 3.07 GHz Intel i7 processor, 16 GB of RAM, 500 GB of hard disk space that was running Ubuntu 10.10.

Unless otherwise noted, no algorithms used more that one core of processor. Consequently, other tasks, such as operating system functions, did not have any impact on the running time of the results.

Table 3.1

A description of all datasets used for evaluation in this dissertation.

UCI Dataset Name	Short Name	n	N
Adult	Adult	14	30,162
Congressional Voting Records	Voting	17	435
Letter Recognition	Letter	17	20,000
Statlog (Vehicle Silhouettes)	Statlog	19	752
Hepatitis	Hepatitis	20	126
Image Segmentation	Image	20	2,310
Imports	Imports	22	205
SPECT Heart	Heart	23	267
Mushroom	Mushroom	23	8,124
Parkinsons	Parkinsons	23	195
Wall-Following Robot Navigation Data	Robot	25	5,456
Automobile	Auto	26	159
Horse Colic	Horse	28	300
Steel Plates Faults	Steel	28	1,941
Flags	Flags	29	194
Breast Cancer Wisconsin (Diagnostic)	WDBC	31	569
Soybean (Large)	Soybean	36	266
Alarm*	Alarm	37	1,000
Water Treatment Plant	Water	38	380
Cylinder Bands	Bands	39	277
SPECTF Heart	SPECTF	45	267
Lung Cancer	Lung	57	26

UCI Dataset Name gives the full name of the dataset in the UCI machine learning repository. *Short Name* gives the name by which the dataset is refered in the later evaluation sections. *n* gives the number of variables in the dataset. *N* gives the number of records in the dataset.

* Alarm is not a dataset from UCI. Gibbs sampling was used to generate a dataset from the Alarm network in the Bayesian Network Repository (http://www.cs.huji.ac.il/site/labs/compbio/Repository/).

Table 3.2

A description of all external algorithm implementations used in this evaluation.

Algorithm	Abbr	Ref	URL
Dynamic Programming	DP	[82]	http://www.cs.helsinki.f /u/tsilande/sw/bene/
Branch and Bound	BB	[19]	http://www.ecse.rpi.edu/ cvrl/structlearning.html
Optimal Reinsertion	OR	[64]	http://www.autonlab.org/autonweb/10530.html

Algorithm gives the name of the algorithm. *Abbr* gives the abbreviation used to refer to this implementation in the results sections in later chapters. *Ref* gives the paper associated with the implementation. *URL* gives the URL from which the code can be downloaded.

CHAPTER 4

HEURISTIC GRAPH SEARCH PERSPECTIVE

As described in Chapter 2, dynamic programming calculates three main functions: $Score(\mathbf{U})$, $BestScore(X, \mathbf{U})$ and $Score(X|\mathbf{U})$. This work addresses the memory bottleneck of current dynamic programming algorithms by considering the problem as a series of graph search problems. This perspective allows adaptation of memory-efficient heuristic search techniques to find optimal structures. As shown in later chapters, comparison to current state of the art dynamic programming techniques show that heuristic search techniques typically run several times faster and use much less memory. This chapter presents the heuristic search perspective of the problem. First, we describe the *order graph*, which is analogous to the dynamic programming lattice. We next define several auxiliary data structures used during the search. *Parent graphs* aid in calculating the cost of edges in the order graph; we give two formulations of parent graphs. Finally, we precompute and cache all necessary scores at the beginning of all of our search algorithms using a strategy similar to that of an AD-tree [62].

4.1 Learning Optimal Subnetworks

Like the dynamic programming algorithms described in Section 2.3.4.4, we find the optimal Bayesian network structure for all variables by learning optimal networks over

subsets of variables. We use an order graph to learn the optimal subnetworks. In some of our algorithms, we begin the search with no variables. For search in this direction, we use a *forward order graph*. Other algorithms begin the search with all of the variables. We use a *reverse order graph* for searches beginning with all of the variables.

4.1.1 Forward Order Graph

Figure 4.1 displays a forward order graph for four variables. It contains subsets of all variables, so the order graph has 2^n nodes. The top-most node in layer 0 containing no variables is the *start node*. The bottom-most node containing all variables is the *goal node*. A directed path in the order graph from the start node to any other node induces an ordering on the variables in the path with new variables appearing later in the ordering. For example, the path traversing nodes $\emptyset, \{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\}$ stands for the variable ordering X_1, X_2, X_3. Each edge on the path has a cost equal to $BestScore$ for the new variable in the child node given the variables in the parent node as candidate parents. For example, the edge between $\{X_1, X_2\}$ and $\{X_1, X_2, X_3\}$ has a cost equal to $BestScore(X_3, \{X_1, X_2\})$. Each order node contains information including a subset of variables, the cost of the best path from the start node to this node, a leaf variable and its optimal parent set. The shortest paths from the start node to all the other nodes correspond to the optimal subnetworks, among which the shortest path to the goal node corresponds to a f nal optimal Bayesian network. The lattice divides the nodes into layers. Nodes in layer l contain optimal subnetworks of l variables. Layer l has $C(n, l)$ nodes, where $C(n, k)$ is the binomial coeff cient.

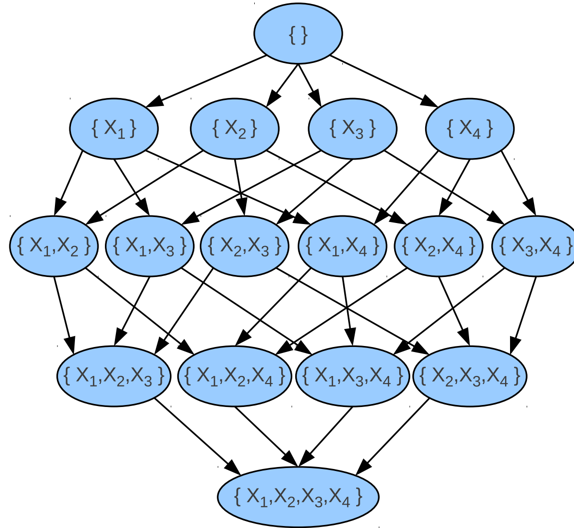


Figure 4.1

A forward order graph of four variables.

4.1.2 Reverse Order Graph

Figure 4.2 displays a reverse order graph for four variables. It is similar to the forward order graph; however, the top-most, start node contains all variables, while the bottom-most, goal node contains none of the variables. A directed path again corresponds to an ordering on the variables: the ordering is the reverse of the order in which the leaves were removed. For example, the path traversing nodes $\{X_1, X_2, X_3, X_4\}$,$\{X_1, X_2, X_3\}$,$\{X_1, X_2\}$, $\{X_1\}$,$\emptyset$ corresponds to the reverse of the variable ordering X_4, X_3, X_2, X_1, which is X_1, X_2, X_3, X_4. An edge between node $\mathbf{U}$ and $\mathbf{U} \setminus \{X\}$ has a cost equal to $BestScore(X, \mathbf{U} \setminus \{X\})$. The shortest path between the start node and the goal node again corresponds to the optimal Bayesian network. Intuitively, the forward order graph adds leaves one at a time, and the candidate parent set for a node is all variables that have been added in the path from the start node to that node. In constrast, the reverse order graph removes leaves one at

a time. The candidate parent set for a node in the reverse order graph is all variables which have not been removed in the path from the start node to that node. Consequently, nodes at shallow layers in the forward order graph have small candidate parent sets, but shallow nodes in the reverse order graph have large candidate parent sets. Similarly, deeper nodes in the forward order graph can select from large candidate parent sets, and the candidate parent sets for deep nodes in the reverse order graph are more restricted.

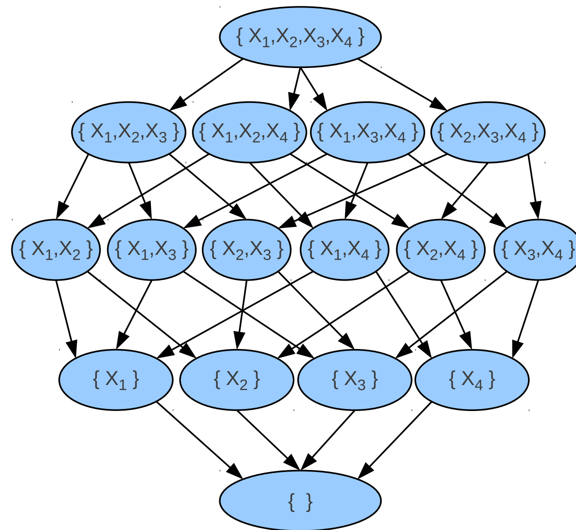


Figure 4.2

A reverse order graph of four variables.

4.2 Identifying Optimal Parent Sets

In order to expand nodes in the order graphs, we need the $BestScore(X_i|\cdot)$ values. We calculate those values using *parent graphs*. Each variable has its own parent graph. We consider two different implementations of parent graphs. The f rst *full parent graphs* mirror

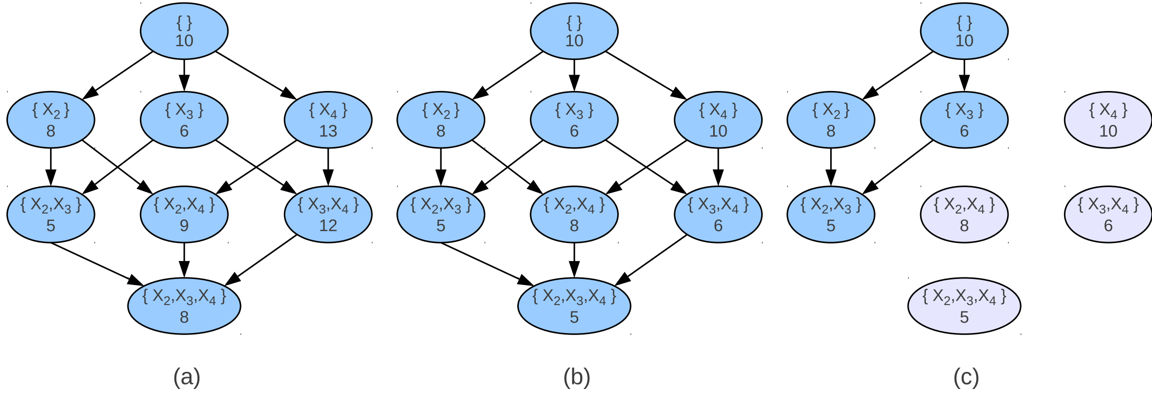
the structure of the order graph. *Sparse parent graphs* adopt a different data structure that often offers considerable memory savings in practice.

4.2.1 Full Parent Graphs

Figure 4.3 shows the construction of the full parent graph for variable X_1 as a lattice. All 2^{n-1} subsets of all other variables are present in the graph. Each node contains one value for $BestScore$ of X_1 and the set of candidate parents shown. That is, each node stores the subset of parents from the given candidate set which minimizes the score of X_1. In Figure 4.3(a), we show the score of X_1 using the indicated set of parents. Figure 4.3(b) shows the f nal parent graph in which each has the optimal set of parents for that candidate parent set. The score of that conf guration is also stored. As with the order graph, the lattice divides the nodes into layers. We call the f rst layer of the graph, the layer with the single node for $\emptyset$ in Figure 4.3, layer 0. A node in layer l has l predecessors, all in layer $l-1$. Layer l has $C(n-1, l)$ nodes. Thus, in total, the complete set of parent graphs stores $n2^{n-1}$ optimal parent sets.

4.2.2 Sparse Parent Graphs

The full parent graph for a variable X exhaustively enumerates all subsets of $\mathbf{V} \setminus \{X\}$ and stores $BestScore(X, \mathbf{U})$ for each subset $\mathbf{U}$. Naively, this approach requires storing $n2^{n-1}$ scores and parent sets [82]. A memory-eff cient approach described in Chapter 5 reduces the memory requirement by storing only one layer of each parent graph in memory at once. That still requires storing $O(nC(n-1, \frac{n}{2}))$ scores, though. This much information is stored in order to make retrieving the optimal parent sets eff cient (i.e., they are stored



(a) The raw scores for all the parent sets. The first line in each node gives the parent set, and the second line gives the score of using all of that set as the parents for X_1. (b) The optimal scores for each candidate parent set. The second line in each node gives the optimal score using some subset of the variables in the first line as parents for X_1. (c) The unique optimal parent sets and their scores. The pruned parent sets are shown in gray. A parent set is pruned if any of its predecessors has a better score.

Figure 4.3

A sample parent graph for variable X_1.

in a hash table or similar data structure). However, the number of *unique* optimal parent sets is often far smaller than either of these numbers because the same parent set is often optimal for many candidate parents sets as described by the following theorem [89].

Theorem 1 *Let* $\mathbf{U} \subset \mathbf{T}$ *and* $X \notin \mathbf{T}$*. If* $Score(X|\mathbf{U}) < Score(X|\mathbf{T})$*,* $\mathbf{T}$ *cannot be the optimal parent set for* X*.*

For example, Figure 4.3(b) shows that a score may be shared by several nodes in a parent graph. The full parent graph representation allocates space for this repetitive information for each candidate parent set, resulting in waste of space.

4.2.2.1 Construction

Sparse parent graphs adopt a different approach. We f rst calculate scores (see Section 4.3) and prune according to Theorem 1. We next *sort* all the unique parent scores

for each variable X in a list, and also maintain a parallel list that stores the associated optimal parent sets. Table 4.1 shows the sorted lists for the parent graph in Figure 4.3(b). We call these sorted lists $scores_X$ and $parents_X$. If we allow X to use all the other variables as candidate parents, then $BestScore(X, \mathbf{V} \setminus \{X\})$ is simply the first element in the sorted list. For example, the first score in Table 4.1 is optimal for the candidate parent set $\{X_2, X_3, X_4\}$. Suppose we remove X_2 from consideration as a candidate parent. We can scan the list from the beginning. As we scan each score, we check the associated parent set. As soon as we find a parent set which does not include X_2, we have found $BestScore(X_1, \{X_3, X_4\})$. Similarly, if we remove both X_2 and X_3, we scan until finding a parent set which includes neither X_2 nor X_3; that is $BestScore(X_1, \{X_4\})$. In essence, this allows us to store and efficiently process only scores in Figure 4.3(c); suboptimal parent sets are never stored or processed, as shown in Table 4.1.

Because of the pruning of suboptimal scores, this approach requires less memory than storing all the possible parent sets and scores. As long as $\|scores_X\| < C(n-1, \frac{n}{2})$, it also requires less memory than the more memory-efficient algorithm for X. In practice, $\|scores_X\|$ is almost always smaller than $C(n-1, \frac{n}{2})$ by several orders of magnitude. So this approach offers (usually substantial) memory savings compared to previous best approaches. However, searching the lists to find optimal parent sets can be inefficient if not done properly. Since we have to search for each arc, the inefficiency of the searching can have a large impact on the the whole search algorithm.

4.2.2.2 Eff cient Scanning

We propose the following eff cient scanning technique. The basic idea is to f rst allow all variables to be candidate parents and successively remove one variable at a time from the candidate parent set. For each variable X, we f rst initialize a working bit vector of length $\|scores_X\|$ called $valid_X$ to be all 1s. This indicates that all the parent scores in $scores_X$ are usable. Therefore, the f rst score in the list will be the optimal score. Then, we create $n-1$ bit vectors also of length $\|scores_X\|$, one for each variable in $\mathbf{V} \setminus \{X\}$. The bit vector for variable Y is denoted as $parents_X^Y$ and contains 1s for all the parent sets that contain Y and 0s for others. Table 4.2 shows the bit vectors for the example in Table 4.1. Then, to exclude variable Y as a candidate parent, we perform the bit operation $valid_X^{new} \leftarrow valid_X \& \sim parents_X^Y$. The new $valid_X$ bit vector now contains 1s for all the parent sets that are subsets of $\mathbf{V} \setminus \{Y\}$. The *f rst set bit* corresponds to $BestScore(X, \mathbf{V} \setminus \{Y\})$. Table 4.3 shows an example of excluding X_3 from the set of possible parents for X_1, and the f rst set bit in the new bit vector corresponds to $BestScore(X_1, \mathbf{V} \setminus \{X_3\})$. If we further want to exclude X_2 as a candidate parent, the new bit vector from the last step becomes the current bit vector for this step, and the same bit operation is applied: $valid_X^{new} \leftarrow valid_X \& \sim parents_{X_1}^{X_2}$. The f rst set bit of the result corresponds to $BestScore(X_1, \mathbf{V} \setminus \{X_2, X_3\})$. Table 4.4 demonstrates this operation. These operations give rise to the $calculateBestScore$ and $createSparseParentGraph$ procedures in Table 4.5. Also, it is important to note that we exclude one variable at a time. For example, if, after excluding X_3, we wanted to exclude X_4 rather than X_2, we

could take $valid_X^{new} \leftarrow valid_X \& \sim parents_{X_1}^{X_4})$. In total, we store $scores_X$ and $parents_X$ for each X and $\sim parents_X(Y)$ for each X and Y.

Table 4.1

Sorted scores and parent sets for X_1 after pruning parent sets which are not possibly optimal.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
$scores_{X_1}$	5	6	8	10

4.2.2.3 Memory Savings

We evaluated the memory savings made possible by using our sparse representation in comparison to the full parent graph data structures. In particular, we compared the maximum number of scores that have to be stored for all variables at once by each algorithm. A typical dynamic programming algorithm stores scores for all possible parent sets of all variables. Memory-eff cient dynamic programming [59] (assuming implementation optimizations) and an algorithm described in Chapter 5 store all possible parent sets only in one layer of the parent graphs for all variables. The sparse representation requires the unique optimal parent sets for all variables at all layers in the search.

As Figure 4.4 shows, the memory savings due to the pruning of provably suboptimal scores is signif cant. In fact, the number of unique scores is typically several orders of magnitude smaller than the number of parent sets stored by the other approaches. These results agree with previously published results [18]. Here, we conf rm that the trend of

Table 4.2

The $parents_X(X_i)$ bit vectors for X_1.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
X_2	1	0	1	0
X_3	1	1	0	0
X_4	0	0	0	0

A "1" in line X_i indicates that the corresponding parent set includes variable X_i, while a "0" indicates otherwise. Note that, after pruning, none of the optimal parent sets include X_4.

Table 4.3

The result of performing the bitwise operation to exclude all parent sets which include X_3.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
$valid_{X_1}$	1	1	1	1
$\sim X_3$	0	0	1	1
$valid_{X_1}^{new}$	0	0	1	1

A "1" in the $valid_{X_1}$ bit vector means that the parent set does not include X_3 and can be used for selecting the optimal parents. The f rst set bit indicates the best possible score and parent set.

Table 4.4

The result of performing the bitwise operation to exclude all parent sets which include either X_3 or X_2.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
$valid_{X_1}$	0	0	1	1
$\sim X_2$	0	1	0	1
$valid_{X_1}^{new}$	0	0	0	1

A "1" in the $valid_{X_1}^{new}$ bit vector means that the parent set includes neither X_2 nor X_3. The initial $valid_{X_1}$ bit vector had already excluded X_3, so f nding $valid_{X_1}^{new}$ only required excluding X_2.

Table 4.5

Sparse parent graph algorithms.

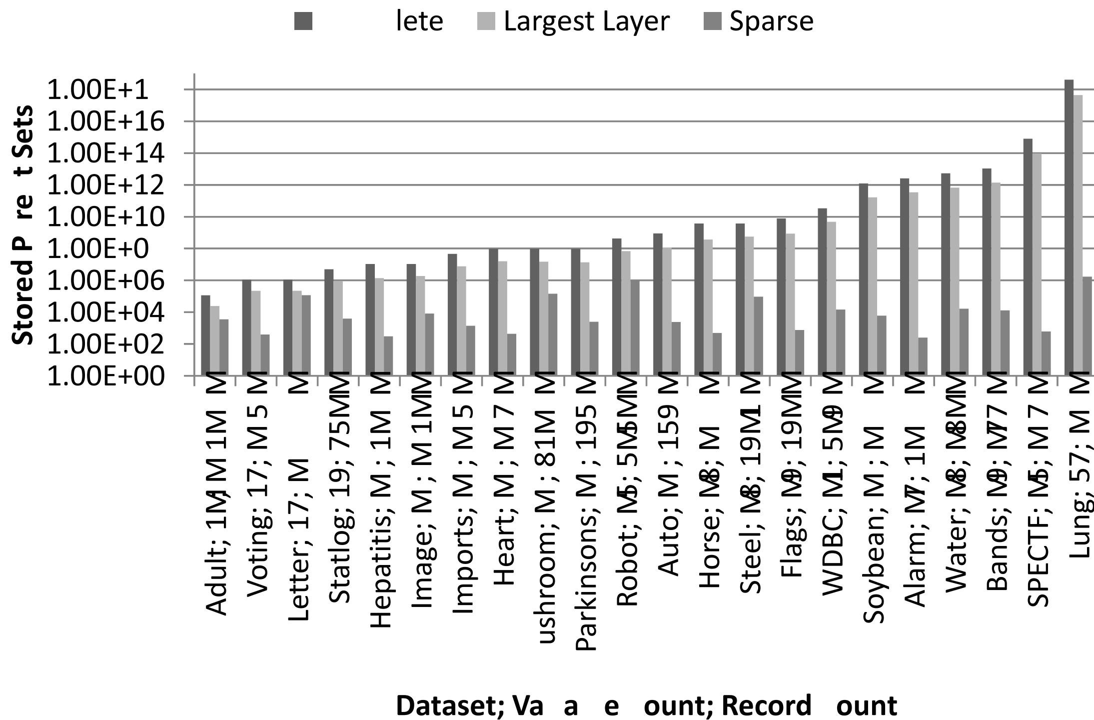
```
1: procedure CALCULATEBESTSCORE(X, U)
2:     valid ← allScores_X
3:     for each Y ∈ V \ U do
4:         valid ← valid& ∼ parents_X^Y
5:     end for
6:     fsb ← firstSetBit(valid)
7:     return scores_X[fsb]
8: end procedure

9: procedure CREATESPARSEPARENTGRAPH(X)
10:     scores_X, parents_X ←sort(Scores(X|·))
11:     for i = 0 → |scores_X| do
12:         for each Y ∈ parents_X(i) do
13:             set(parents_X^Y(i))
14:         end for
15:     end for
16: end procedure
```

sparse parent graph size independent from n on datasets up to size 57. BB and mathematical programming enjoy similar memory savings because they only include unique optimal parent sets, as well. However, the nature of those searches does not suggest a clear relationship between pruning scores and memory savings during the execution of their algorithm. Figure 4.4 also suggests that the savings increase as the number of variables increases in the datasets. This implies that, while more variables necessarily increases the number of candidate parent sets exponentially, the number of unique optimal parent sets increases much more slowly. Intuitively, even though we add more possible parents, only a small number of them are "good" parents for any particular variable.

For most of our algorithms, we present results using both the full and the sparse representations of parent graphs. All of the sparse versions benef t similarly from the reduced memory, so we do not repeat those results in each section. In most cases, the sparse parent graphs also yield signif cant runtime improvements. Because those results vary from algorithm to algorithm, they are discussed in more detail in the appropriate chapters.

4.3 Calculating Scores

We use an AD-tree-like search to calculate all of the parent scores. An AD-tree [62] is an unbalanced tree which contains AD-tree nodes and varying nodes. The tree is used to collect count statistics from a dataset. An AD-node stores the number of records consistent with the variable instantiation of the node, while a varying node assigns a value to a variable. Figure 4.5 shows an AD-tree. As described in Chapter 2 and shown in Equation 4.2, the scores can be calculated based on parent instantiation counts and variable and parent



Complete gives the e of sets sto ed y a typical dyna ic p ogra i g strategy, which is all of the co plete pa e t graphs.

L e t L ye gives the e of sets sto ed y e o y-efficie t dyna ic progra ing a d the algo ith desc ibed i Chapte 5, which is the la gest laye s of all pa e t graphs.

Spa e gives the e of sets sto ed y the spa se pa e t graphs, which is the sets which a e ot p ned d e to Theo e 1.

Figure 4.4

The maximum count of parent sets stored by each of the parent graph strategies.

instantiation counts. A count statistic is only used once and can be discarded afterwards. Therefore, we can use a depth-first traversal of the AD-tree to compute the parent scores to minimize the search space needed. In particular, our implementation calculates the MDL score. In addition to Theorem 1, we also use the following theorem [90] to prune the tree.

Theorem 2 *In an optimal Bayesian network based on the MDL scoring function, each variable has at most* $\log(\frac{2N}{\log N})$ *parents, where* N *is the number of data points.*

The theorem states that only small parent sets can possibly be optimal parents when using the MDL score. All nodes below the depth specified in the theorem can be pruned without computing them.

While Equation 2.6 does accurately express the MDL scoring function, it is not particularly amenable to efficient calculation. Consequently, we use the following (equivalent) equation for calculating the MDL score.

$$MDL(G) = \sum_i MDL(X_i|PA_i), \tag{4.1}$$

where

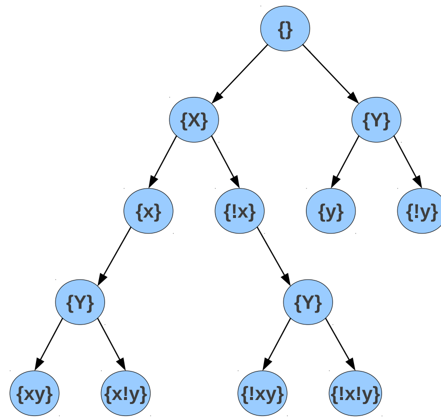
$$MDL(X_i|PA_i) = H(X_i|PA_i) + \frac{\log N}{2} K(X_i|PA_i), \tag{4.2}$$

$$H(X_i|PA_i) = -\sum_{x_i, pa_i} N_{x_i, pa_i} \log \frac{N_{x_i, pa_i}}{N_{pa_i}}, \tag{4.3}$$

$$= \sum_{x_i, pa_i} N_{x_i, pa_i} \log N_{pa_i} - \sum_{x_i, pa_i} N_{x_i, pa_i} \log N_{x_i, pa_i}, \tag{4.4}$$

$$K(X_i|PA_i) = (r_i - 1) \prod_{X_l \in PA_i} r_l. \tag{4.5}$$

Table 4.6 gives an algorithm which eff ciently implements these equations. All of our algorithms precompute the score cache using this algorithm.



The AD-nodes have lower case letters which show the instantiation of individual variables. For example, the label *x!y* means that *x* is *true* and *y* is *false*. Vary nodes have an upper case letter and show which variable was varied.

Figure 4.5

An AD-tree.

Table 4.6

Score calculation algorithm.

1: **procedure** CALCULATESCORES($\mathbf{D}$)
2: updateScores($\emptyset, \mathbf{D}$)
3: expandADNode($X_{-1}, \emptyset, \mathbf{D}$)
4: **for** each $X \in \mathbf{V}$ **do**
5: prune($X, \emptyset, Score(X|\emptyset)$)
6: **end for**
7: **end procedure**

8: **procedure** EXPANDADNODE($X_i, \mathbf{U}, \mathbf{D}_u$)
9: **For** $j = i + 1 \rightarrow n$ **do** expandVaryNode($j, \mathbf{U}, \mathbf{D}_u, d$)
10: **end procedure**

11: **procedure** EXPANDVARYNODE($X_i, \mathbf{U}, \mathbf{D}_u$)
12: **for** $j = 0 \rightarrow r_i$ **do**
13: updateScores($\mathbf{U} \cup \{X_i\}, \mathbf{D}_{X_i=j,u}$)
14: **if** $|\mathbf{U}| < \log(\frac{2N}{\log N})$ **then** expandADNode($i, \mathbf{U} \cup \{X_i\}, \mathbf{D}_{X_i=j,u}$)
15: **end for**
16: **end procedure**

17: **procedure** UPDATESCORES($\mathbf{U}, \mathbf{D}_u$)
18: **for** each $X \in \mathbf{V} \setminus \mathbf{U}$ **do**
19: **if** $Score(X|\mathbf{U})$ is null **then** $Score(X|\mathbf{U}) \leftarrow K(X|\mathbf{U})$
20: $Score(X|\mathbf{U}) \leftarrow Score(X|\mathbf{U}) + |\mathbf{D}_u| * \log |\mathbf{D}_u|$
21: **end for**
22: **for** each $X \in \mathbf{U}$ **do**
23: $Score(X|\mathbf{U} \setminus \{X\})$ is null $Score(X|\mathbf{U} \setminus \{X\}) \leftarrow K(X|\mathbf{U} \setminus \{X\})$
24: $Score(X|\mathbf{U} \setminus \{X\}) \leftarrow Score(X|\mathbf{U} \setminus \{X\}) - |\mathbf{D}_u| * \log |\mathbf{D}_u|$
25: **end for**
26: **end procedure**

27: **procedure** PRUNE($Y, \mathbf{U}, bestScore$)
28: **for** each $X \in \mathbf{V} \setminus \mathbf{U}$ **do**
29: **if** $Score(X|\mathbf{U} \cup X) < bestScore$ **then**
30: prune($Y, \mathbf{U} \cup X, Score(X|\mathbf{U} \cup X)$)
31: **else**
32: delete $Score(X|\mathbf{U} \cup X)$
33: prune($Y, \mathbf{U} \cup X, bestScore$)
34: **end if**
35: **end for**
36: **end procedure**

CHAPTER 5

BEST-FIRST HEURISTIC SEARCH

A* [42] is a best-ﬁrst heuristic graph search algorithm. It requires an evaluation function $f(n)$ and a successor operator $\Gamma(n)$. The evaluation function for a node n consists of two terms: the cost so far, $g(n)$, and the *admissible* cost from n to a goal node, $h(n)$. An admissible cost is an optimistic estimate of the distance from n to a goal node; that is, the admissible cost is an underestimate, or lower bound, on the distance from n to the goal. If the heuristic function $h(n)$ is *consistent*, A* guarantees to ﬁnd the shortest path from the start node to each node expanded. A consistent function always underestimates the cost from n to a goal node and assigns the same value or a higher value to all successors of n. Consequently, a consistent function is always admissible. Application of the successor operator to node n, *expanding* n, returns the successors of n as well as the cost from n to each successor. The algorithm begins by placing the start node on a priority queue called $openlist$. The open list is implemented as a heap in which nodes are organized according to increasing f values. At each iteration, the head of $openlist$ is expanded and placed in a $closedlist$. The closed list is implemented as a hash table. For the Bayesian network structure learning problem, the key of the hash table is a subset of variables and the value is the node which corresponds to that subset. The f cost of each successor is calculated. Duplicate detection is performed by checking $open$ and $closedlists$ for each successor. If

Table 5.1

A* search algorithm.

```
1: procedure MAIN(D)
2:     calculateScores(D)
3:     for each X ∈ V do
4:         BestScore(X, ∅) ← Score(X|∅)
5:         calculateParentGraph(X, ∅)
6:     end for
7:     push(open, φ, ∑_{Y∈V} BestScore(Y, V \ {Y})
8:     while !isEmpty(open) do
9:         U ←pop(open)
10:        if U is goal then
11:            print("The best score is " + Score(V))
12:            return
13:        end if
14:        put(closed, U)
15:        for each X ∈ V \ U do
16:            if contains(closed, U ∪ {X}) then
17:                continue
18:            end if
19:            g ← BestScore(X, U) + Score(U)
20:            h ← ∑_{Y∈V\U} BestScore(Y, V \ {Y})
21:            if g + h < Score(U ∪ {X} then
22:                push(open, U ∪ {X}, g + h)
23:                Score(U ∪ {X}) ← g + h
24:            end if
25:        end for
26:    end while
27: end procedure

28: procedure CALCULATEPARENTGRAPH(Y, U)
29:     for each X ∈ V \ U do
30:         if Score(Y|U ∪ {X}) < BestScore(Y, U)
31:           and Score(Y|U ∪ {X}) < BestScore(Y, U ∪ {X}) then
32:             BestScore(Y|U ∪ {X}) ← Score(Y|U ∪ {X})
33:         else if BestScore(Y|U ∪ {X}) < BestScore(Y, U ∪ {X}) then
34:             BestScore(Y|U ∪ {X}) ← Score(Y|U ∪ {X})
35:         end if
36:         calculateParentGraph(Y, U ∪ {X})
37:     end for
38: end procedure
```

neither data structure contains the successor, it is added to $openlist$ with the calculated f cost and a parent pointer to n. If a successor is on the $openlist$ list with a higher g value, it is updated with the new value and parent pointer. If the successor is in $closedlist$ with a higher g value, it is moved back to $openlist$. Otherwise, the successor is discarded. Once a goal node is selected for expansion, a shortest path is found by following parent pointers backward to the start node.

Table 5.1 gives pseudocode for our A* algorithm to learn optimal Bayesian network structures. Our formulation of the order graph allows us to specify an evaluation function and a successor operator. This algorithm uses a forward order graph. As presented, the algorithm uses the full parent graph representation. The complete parent graphs are constructed before searching through the order graph in the $calculateParentGraph$ procedure. We can easily adapt the algorithm to use sparse parent graphs, though, by replacing the calls to $BestScore(\cdot)$ with calls to $calculateBestScore(\cdot)$ from Table 4.5. If we use sparse parent graphs, then we do not need to use the $calculateParentGraph$ procedure. Instead, we use $createSparseParentGraph$ from Table 4.5 to construct the sparse parent graphs at the beginning of the search.

5.1 Heuristic Function

The best-f rst algorithm def nes $g(n)$ as the sum of edge costs from the start node to n. Each edge cost is $BestScore(X, \mathbf{U})$ where X is the variable added to the ordering. We use the following heuristic function [102],

Def nition 1

$$h(\mathbf{U}) = \sum_{Y \in \mathbf{V} \setminus \mathbf{U}} BestScore(Y, \mathbf{V} \setminus \{Y\}). \tag{5.1}$$

The heuristic function h allows variables missing from the ordering to choose optimal parents from all variables in $\mathbf{V}$. This effectively relaxes the acyclic constraint to quickly calculate a lower bound. The following theorem [102] proves h is consistent. Consistent functions are also admissible.

Theorem 3 *h is consistent.*

Proof: For any successor node $\mathbf{R}$ of $\mathbf{U}$, let $Y \in \mathbf{R} \setminus \mathbf{U}$. We have

$$\begin{aligned} h(\mathbf{U}) &= \sum_{X_i \in \mathbf{V} \setminus \mathbf{U}} BestScore(X_i, \mathbf{V} \setminus \{X_i\}) \\ &\leq \sum_{X_i \in \mathbf{V} \setminus \mathbf{U}, X_i \neq Y} BestScore(X_i, \mathbf{V} \setminus \{X_i\}) + BestScore(Y, \mathbf{U}) \\ &= h(\mathbf{R}) + c(\mathbf{U}, \mathbf{R}). \end{aligned}$$

The inequality holds because fewer variables are used to select optimal parents for Y. Hence, h is consistent. □

5.2 Successor Operator

The forward order graph also suggests the successor operator. To expand node $\mathbf{U}$, we try each $X \in \mathbf{V} \setminus \mathbf{U}$ as a leaf for $\mathbf{U}$. The edge cost is $BestScore(X, \mathbf{U})$, and the g value of the successor is equal to the g value of the predecessor summed with $BestScore(X, \mathbf{U})$.

5.3 Solution Reconstruction

For conciseness, Table 5.1 only includes the main logic in computing the optimal score; however, in addition to storing the optimal score over the variables in $\mathbf{U}$, we also store the leaf and parents which give that score in $leaf(\mathbf{U})$ and $parents(\mathbf{U})$ while a node is in the open list. After expanding a node, we write that information to disk in order to conserve RAM. We also maintain an entry in the closed list that the node has already been expanded. We reconstruct the optimal solution by beginning with the goal node (so $\mathbf{U} = \mathbf{V}$). We consult $leaf(\mathbf{U})$ and $parents(\mathbf{U})$ to find the last leaf, l, and its optimal parent set. We then recursively look up $leaf(\mathbf{U} \setminus \{l\})$ and $parents(\mathbf{U} \setminus \{l\})$ until reconstructing the entire network.

5.4 Advantages of A*

A* offers several advantages over dynamic programming. Primarily, by expanding nodes according to their f values, A* never expands subnetworks which provably cannot compose an optimal complete structure. Subnetworks with a worse f value than the optimal network are either never generated or remain in the open list. The savings typically manifest in both reduced memory complexity, because the unexpanded nodes are not stored in the closed list, and reduced time complexity, because no time is spent expanding those nodes. In some cases, though, the overhead of maintaining the priority queue overshadows the savings from pruning. In these cases, very little pruning occurred, and nearly the entire order graph was expanded.

5.5 Empirical Results

We evaluated our A* search algorithm on a set of benchmark datasets from the UCI repository [33]. We compared to dynamic programming (DP) and branch and bound (BB). We tested using both sparse and full parent graph representations.

Figure 5.1 reports the running time of the three algorithms in solving the benchmark datasets. We terminated an algorithm if it ran for more than 2 hours on a dataset. We also report the sizes of the sparse parent graphs compared to the full parent graphs. Finally, we give the number of expanded nodes for both A* and DP. The difference in sizes demonstrates the computation wasted by dynamic programming evaluating subnetworks which could not possibly compose an optimal structure. We had no way of tracking the size of the search space by BB because only binary code is provided.

The timing results show that our A* algorithm with full parent graphs is typically several times faster than DP and orders of magnitude faster than BB on most of the datasets we tested. A* is slower than DP on $Adult$ and $Letter$, which have a large number of records and a relatively small number of variables, which makes the pruning technique in Theorem 1 less effective. Although the DP algorithm does not perform any pruning, due to its simplicity, the algorithm is highly optimized. Consequently, it was faster than A* search on the $Adult$ and $Letter$ datasets; however, on the $Mushroom$ dataset, which also included a large number of samples but had a larger number of variables, A* runs faster than DP. Because of the exponential size of the parent and order graphs, as the number of variables grows, the amount of pruning of Theorem 1 has less impact on the running time.

The results call particular attention to the benef t of the sparse parent graphs. Because they do not require the construction overhead of the full parent graphs, the runtimes of all of the algorithms are signif cantly reduced. Of the datasets with more than 20 variables, A* with the sparse parent graphs runs more than an order of magnitude faster than DP on all of them except mushroom. As mentioned above, datasets with many records, such as $Mushroom$, limit the pruning offered by Theorem 1. In these cases, searching for $BestScore(\cdot)$ in the sparse representation takes longer to execute. Even under this less-than-ideal circumstance, though, A* with the sparse parent graphs still runs over 5 times faster than DP. These results also show that the eff ciency of the sparse parent graphs do not depend heavily on n.

The sparse parent graphs have another advantage over the full parent graphs for A* search. As evidenced by the number of nodes expanded by A* compared to the number in the complete order graph, at least in some cases, A* never needs some values from the parent graphs. However, the full parent graphs will calculate these values anyway, since they are created in full at the beginning of the search. On the other hand, the sparse parent graphs do not compute a value until asked, so they never waste time calculating scores that are never necessary.

Finally, BB is much slower than A* with or without the sparse parent graphs. Its search space includes graphs with cycles, while the A* search space does not. The results indicate that it is better to search in the space of DAGs directly in f nding an optimal Bayesian network structure.

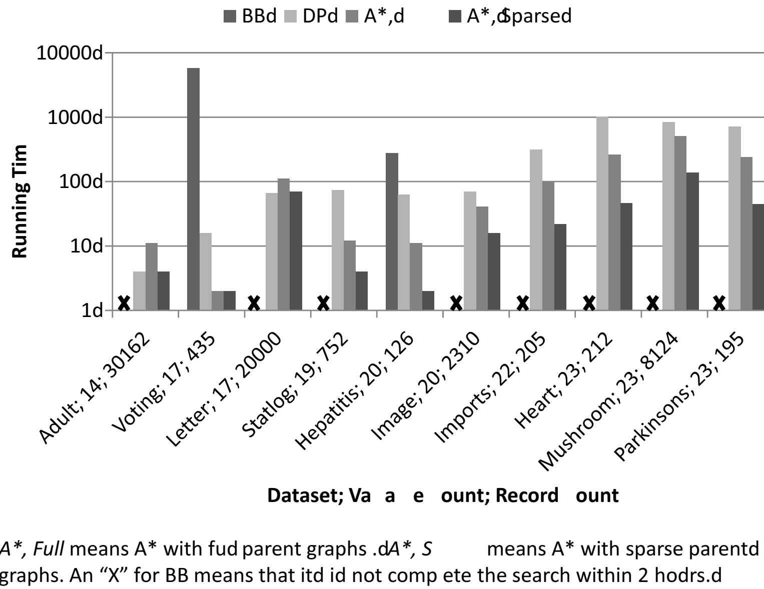


Figure 5.1

Runtime comparison among BB, DP and A*.

Figure 5.2 plots the number of nodes expanded by A* versus the full size of the order graph at each layer for adult and hepatitis. The heuristic function used by A* initially provides only loose bounds, so A* expands most nodes in the beginning layers. The heuristic bounds tighten as the search progresses, so A* prunes more nodes at deeper layers. For the adult dataset, A* expanded almost all the order nodes in the beginning 7 layers of the order graph before it started to prune order nodes in the f nal layers. In contrast, only a small percentage of the nodes were expanded in the order graph of hepatitis. The pruning became quite effective as early as at layer 4 and 5, and only a few nodes were expanded in the last 10 layers.

Figure 5.3 shows the benef t of pruning on all of the datasets. The f gure shows that A* always expands fewer nodes that DP. On some of the datasets, such as $Letter$, the savings

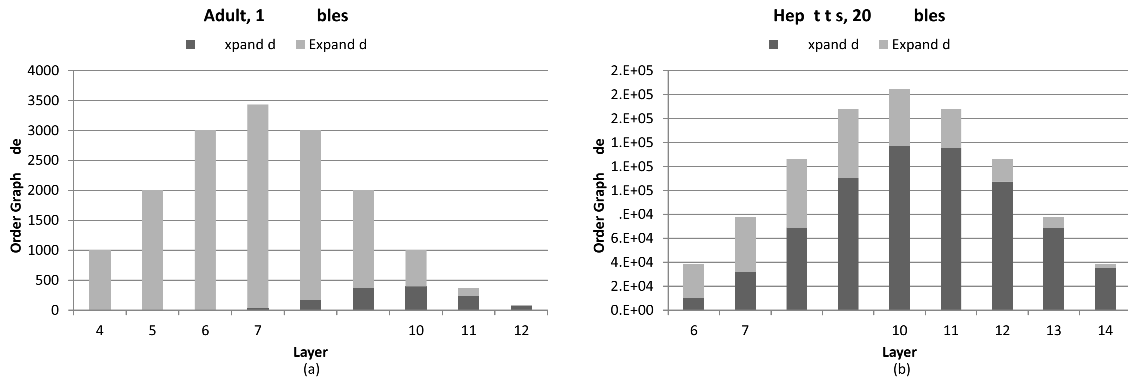


Figure 5.2

Nodes expanded by A* at the middle layer of two datasets.

are modest. For $Voting$, though, A* expands almost an order of magnitude fewer nodes than DP. Unexpanded nodes reduce both memory and runtime costs because no work is wasted storing or processing the unexpanded nodes.

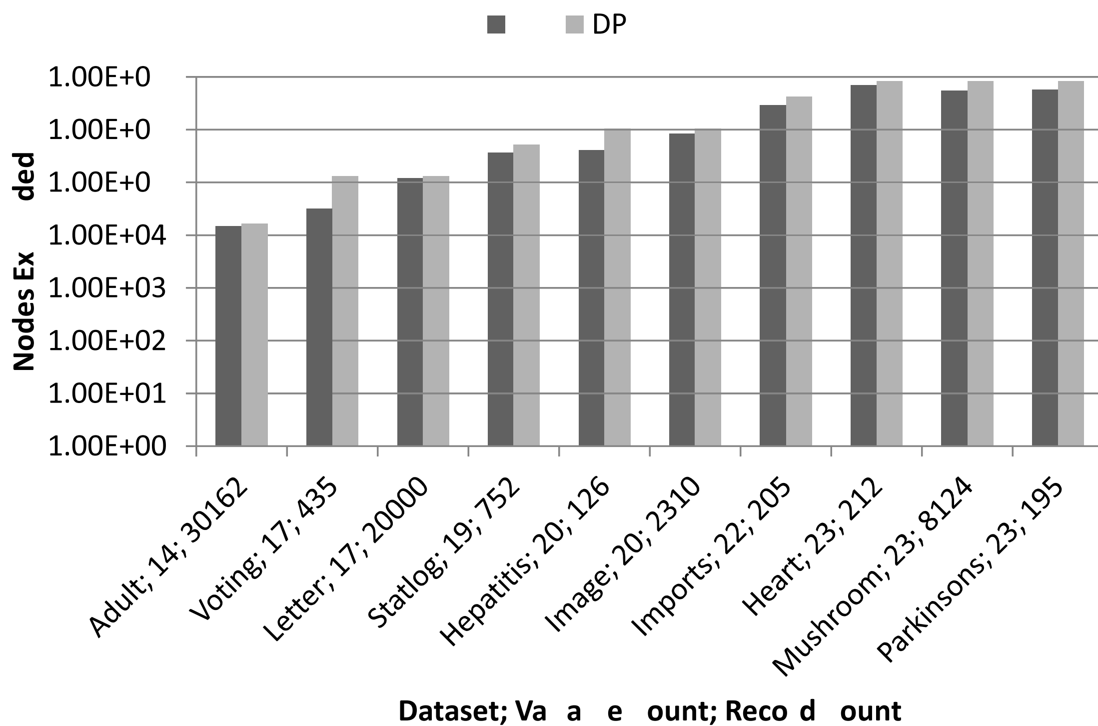


Figure 5.3

Comparison of the order graph nodes expanded by DP and A*.

CHAPTER 6

FRONTIER BREADTH-FIRST BRANCH AND BOUND SEARCH

Our best-first heuristic search results agree with previous results [70] which show that learning optimal Bayesian networks is typically limited by RAM. This was more pronounced when using the full parent graphs, but, especially on datasets for which Theorem 1 was not very effective, the sparse parent graphs combined with the closed list for A* sometimes consumed a sizable amount of RAM. We next attempted to reduce the memory complexity of learning optimal Bayesian networks by taking advantage of the regular structure of the order and parent graphs. In particular, we observed that only a limited amount of information is required to generate each layer of the parent and order graphs. Generating a layer of a parent graph requires the previous layer of that parent graph and corresponding scores. A layer in the order graph requires the previous layer of the order graph and the current layer of the parent graphs. Because scores and parent sets are propagated from layer to layer in the parent graphs, a layer can be deleted once it has been expanded. Similarly, scores in the order graph are propagated; these can also deleted once expanded. Reconstructing the network structure necessitates the leaf, its optimal parent set and a pointer to its predecessor for each order graph node be stored on disk.

To overcome the memory constraint and leverage the structure present within the search space, we introduce a *frontier breadth-first branch and bound algorithm with delayed du-*

Table 6.1

A frontier BFBnB search algorithm.

```
1: procedure EXPANDORDERGRAPH(l, isPresent, upper, maxSize)
2:     for each Score_l(U) ∈ Score_l do
3:         for each X ∈ V \ U do
4:             g ← Score_l(U) + BestScore_l(X|U)
5:             h ← ∑_{Y∈V\U} BestScore(Y, V \ {Y})
6:             if g + h > upper then continue
7:             isPresent(U ∪ {X}) ← true
8:             if g < Score_{l+1}(U ∪ {X}) then
9:                 Score_{l+1}(U ∪ {X}) ← g
10:            end if
11:            if |Score_{l+1}| > maxSize then writeTempFile(Score_{l+1})
12:        end for
13:    end for
14:    Score_{l+1} ← mergeTempFiles; delete Score_l
15: end procedure

16: procedure EXPANDPARENTGRAPH(l, p, isPresent, maxSize)
17:    for each BestScore_l(p|U) ∈ BestScore_l(p do
18:        for each X ∈ V \ U and X ≠ p do
19:            S ← U ∪ {X}
20:            if !isPresent(S) then continue
21:            if Score(p|S) < BestScore_{l+1}(p|S) then
22:                BestScore_{l+1}(p|S) ← Score(p|S)
23:            end if
24:            if BestScore_l(p|U) < BestScore_{l+1}(p|S) then
25:                BestScore_{l+1}(p|S) ← BestScore_l(p|U)
26:            end if
27:            if |BestScore_{l+1}(p)| > maxSize then writeTempFile(BestScore_{l+1}(p))
28:        end for
29:    end for
30:    BestScore_{l+1}(p) ←mergeTempFiles; delete BestScore_l(p)
31: end procedure

32: procedure MAIN(D, upper, maxSize)
33:    calculateScores(D); writeScoresToDisk
34:    for l = 1 → n do
35:        for p = 1 → n do expandParentGraph(l, p, isPresent, maxSize)
36:        expandOrderGraph(l, isPresent, upper, maxSize)
37:    end for
38:    print("The best score is " + Score(V))
39: end procedure
```

plicate detection by adapting the breadth-first heuristic search algorithm proposed by Zhou and Hansen [104, 106]. It is also similar to the frontier search described by Korf [49]. We use the same notation for heuristic search introduced in Chapter 4.

Breadth-first heuristic search expands a search space in order of layers of increasing g-cost with each layer comprising all nodes with a same g-cost. As each node is generated, a heuristic function is used to calculate a lower bound for that node. If the lower bound is worse than a given upper bound on the optimal solution, the node is pruned; otherwise, the node is added to the open list for further search. A divide-and-conquer method is used to reconstruct the optimal solution.

Table 6.1 gives the pseudocode for our BFBnB search algorithm for learning optimal Bayesian networks. Like the A* search, it also uses the forward order graph. The algorithm is similar to the breadth-first heuristic search algorithm but has several differences. First, the layers in our search graphs (the parent and order graphs) do not correspond to the g-costs of nodes; rather, layer l corresponds to variable sets (candidate parent sets or optimal subnetworks) of size l. For the order graph, though, we can calculate both a g- and h-cost for pruning. This pruning can also be propagated to the parent graphs, as described in Section 6.1. Another difference is that, when using the full parent graphs, our search problem is an interlaced search of order and parent graphs which must be carefully orchestrated to ensure the correct nodes can be accessed easily at the correct time, as described in Section 6.2. This further requires the scores are stored in particular order, as described in Section 6.3. Yet another difference is that we use a variant of delayed duplicate detection [51] in which external memory is not used to detect duplicates until the open list

exceeds the size of RAM, as described in Section 6.4. Finally, we use the network structure reconstruction algorithm described in Section 5.3 rather than using divide-and-conquer to reconstruct the optimal solution.

As with the A* search algorithm, Algorithm 6.1 uses full parent graphs. The algorithm can easily be adapted to use sparse parent graphs, though, by constructing them at the beginning of the search and replacing the calls to $BestScore(\cdot)$ with the appropriate procedures from Table 4.5.

The pseudocode only includes the logic to calculate the optimal score. The optimal network is reconstructed using the technique described in Section 5.3.

6.1 Branch and Bound

In order to safely prune nodes, we need a heuristic function $f(\mathbf{U}) = g(\mathbf{U}) + h(\mathbf{U})$ that estimates the cost of the best path from the start node to a goal node using order node $\mathbf{U}$. We use the heuristic function described in Section 5.1.

We also need an upper bound on the score of the optimal Bayesian network in order to prune. A search node $\mathbf{U}$ whose heuristic value $f(\mathbf{U})$ is higher than the upper bound is immediately pruned. Numerous fast, approximate methods exist for learning a locally optimal Bayesian network. We use a greedy hill climbing algorithm with a tabu list and random restarts [40].

6.2 Coordinating the Graph Searches

The parent and order graph searches must be carefully coordinated to ensure that the parent graphs contain the necessary nodes to expand nodes in the order graph. In particular, expanding a node $\mathbf{U}$ in layer l in the order graph requires $BestScore(X, \mathbf{U})$, which is stored in the node $\mathbf{U}$ of the parent graph for X. Hence, before expanding layer $|\mathbf{U}|$ in the order graph, that layer of the parent graphs must already exist. Therefore, the algorithm alternates between expanding layers of the parent graphs and order graph. In both graphs, a hash table is used to detect generated nodes and store their scores.

Expanding a node $\mathbf{U}$ in the parent graph amounts to generating successor nodes with candidate parents $\mathbf{U} \cup \{X\}$ for all X in $\mathbf{V} \setminus \mathbf{U}$. For each successor $\mathbf{S} = \mathbf{U} \cup \{X\}$, the hash table for the next layer is first checked to see if $\mathbf{S}$ has already been generated. If not, the score of using all of $\mathbf{S}$ as parents of X is retrieved from the score cache and compared to the score of using the parents specified in $\mathbf{U}$. If using all of the variables has a better score, then an entry is added to the hash table indicating that, for candidate parents $\mathbf{S}$, using all of them is best. Otherwise, according to Theorem 1, the hash table stores a mapping from $\mathbf{S}$ to the parents in $\mathbf{U}$. Similarly, if $\mathbf{S}$ has already been generated, the score of the existing best parent set for $\mathbf{S}$ is compared to the score using the parents in $\mathbf{U}$. If the score of the parents in $\mathbf{U}$ is better, then the hash table mapping is updated accordingly. Once a layer of the parent graph is expanded, the whole layer can be discarded as it is no longer needed. The pseudocode uses $BestScore_l$ to store the optimal scores at each layer.

Expanding a node $\mathbf{U}$ in the order graph amounts to generating successor nodes $\mathbf{U} \cup \{X\}$ for all X in $\mathbf{V} \setminus \mathbf{U}$. To calculate the score of successor $\mathbf{S} = \mathbf{U} \cup \{X\}$, the score of the

existing node $\mathbf{U}$ is added to $BestScore(X, \mathbf{U})$, which is retrieved from parent graph node $\mathbf{U}$ for variable X. The optimal parent set out of $\mathbf{U}$ is also recorded. This is equivalent to trying X as the leaf and $\mathbf{U}$ as the subnetwork. Next, the hash table for the next layer is consulted. If it contains an entry for $\mathbf{S}$, then a node for this set of variables has already been generated using another variable as the leaf. The score of that node is compared to the score for $\mathbf{S}$. If the new score for $\mathbf{S}$ is better, or the hash table did not contain an entry for $\mathbf{S}$, then the mapping in the hash table is updated. Unlike the parent graph, however, a portion of each order graph node is used to reconstruct the optimal network at the end of the search, as described in Section 5.3. This information is written to disk, while the other information is deleted. The pseudocode uses $Score_l$ to store the score for each subnetwork.

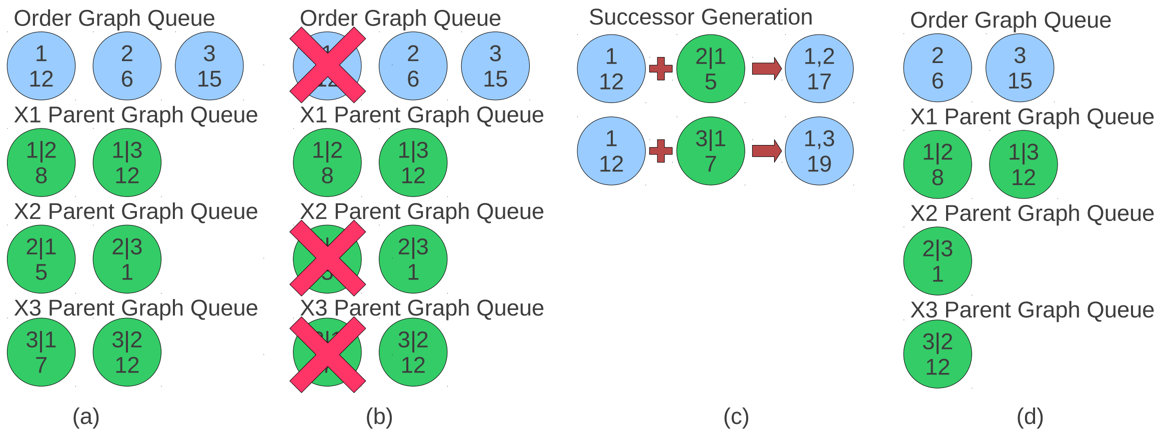
Additional care is needed to ensure that parent and order graph nodes for a particular layer are accessed in a regular, structured pattern. We arrange the nodes in the parent and order graphs in queues such that when node $\mathbf{U}$ is removed from the order graph queue, the head of each parent graph queue for all X in $\mathbf{V} \setminus \mathbf{U}$ is $\mathbf{U}$. So all of the successors of $\mathbf{U}$ can be generated by combining it with the head of each of those parent graph queues. Once the parent graph nodes are used, they can be removed, and the queues will be ready to expand the next node in the order graph queue. Because the nodes are removed from the heads of the queues, these invariants hold throughout the expansion of the layer. Regulating such access patterns improves the scalability of the algorithm because these queues can be stored on disk and accessed sequentially to reduce the requirement of RAM. The regular accesses also reduce disk seek time. The pseudocode assumes the nodes are written to disk in this order to easily retrieve the next necessary node.

The lexicographic ordering [46] of nodes within each layer is one possible ordering that ensures the queues remain synchronized. For example, the lexicographic ordering of 4 variables of size 2 is $\{\{X_1, X_2\}, \{X_1, X_3\}, \{X_2, X_3\}, \{X_1, X_4\}, \{X_2, X_4\}, \{X_3, X_4\}\}$. The order graph queue for layer 2 of a dataset with 4 variables should be arranged in that order. The parent graph queue for variable X should have the same sequence, but without subsets containing X. In the example, the parent graph queue for variable X_1 should be $\{\{X_2, X_3\}, \{X_2, X_4\}, \{X_3, X_4\}\}$. Figure 6.1 shows a simple example of expanding one order graph node by manipulating the necessary queues. As described in more detail in Section 6.4, the nodes of the graphs must be sorted to detect duplicates; the lexicographic order ensures that there is no additional work required to arrange the nodes when writing them to disk.

If the sparse parent graphs are used, there is no coordination problem because each $BestScore(\cdot)$ is caculated from scratch using the eff cient bit-wise operations described in Section 4.2.2.

6.3 Ordering the Scores on Disk

We have assumed that, because of its pruned size due to Theorem 1, the score cache could f t in RAM. For large datasets, though, the score cache can grow quite large. We write it to disk to reduce RAM usage. Each score $Score(X|\mathbf{P})$ is used once, when node $\mathbf{P}$ is f rst generated in the parent graph for X. As described in Section 6.2, the parent graph nodes are expanded in lexicographic order; however, they are not generated in that order. The successors of node $\{X_1\}$ in the parent graph for X_0 are $\{X_1, X_2\}, \{X_1, X_3\}, \{X_1, X_4\}$.... When



(a) The initial nodes in the order and parent graphs. The first line in each order graph node gives the subset of variables for that node. The second line gives the score of the optimal subnetwork for that subset of variables. The first line in each parent graph node gives the variable of the parent graph and the candidate set of parents for that node. The second line shows the score of the optimal parent set for that variable for that set of candidate parents. (b) The first node in each of the appropriate queues is removed. (c) The order graph node is combined with each of the parent graph nodes to generate a new order graph node. (d) The queues are ready for the expansion of the next order graph node.

Figure 6.1

Coordinating the parent and order graphs.

$\{X_2\}$ is expanded, the new successors are $\{X_2, X_3\}, \{X_2, X_4\}$... even though $\{X_2, X_3\}$ precedes $\{X_1, X_4\}$ in lexicographic order. Therefore, the scores must be written in order of successors of nodes expanded in lexicographic order.

A file is created for each variable for each layer to store these sorted scores after all scores not pruned by Theorem 1 are in the score cache. The file for a particular layer can be deleted after expanding that layer in the appropriate parent graph. Each variable set $\mathbf{U}$ is generated in lexicographic order, $\{X_0\}, \{X_1\}, \{X_0, X_1\}, \{X_2\}$.... $\mathbf{U}$ is then expanded as it would be in the parent graphs for variables $\mathbf{V} \setminus \mathbf{U}$. The scores of these successors which had not already been generated are written to disk.

The sparse parent graphs can use external memory sorting during their construction if necessary. In practice, the pruned score cache does not consume more memory than the

order and parent graphs. Also, as demonstrated in Figure 4.4, the size of the cache does not depend on n. For all of the datasets considered in this work, the largest pruned score cache was 20MB (for the 57-variable lung cancer dataset).

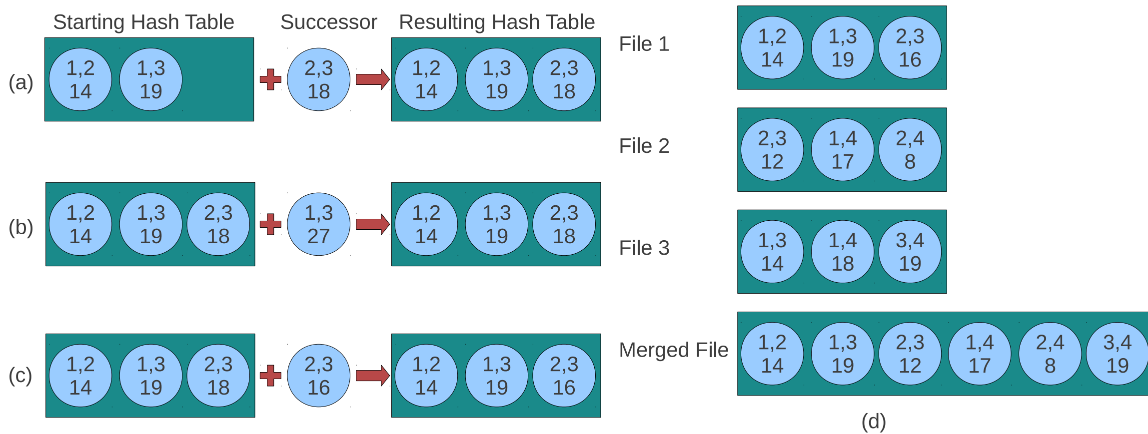
6.4 Duplicate Detection

Duplicate nodes are generated during the parent and order graph searches. Duplicates in the parent and order graphs are nodes which consider the same sets of variables (candidate parent sets and optimal subnetworks, respectively). Because the successors of a node always consider exactly one more variable in both the parent and order graphs, the successors of a node in layer l are always in layer $l + 1$. Therefore, when a node is expanded, its successors could only be duplicates of nodes in the next layer. Duplicates are detected using a hash table in RAM. In both the parent and order graphs, the duplicate with the best score should be kept. After expanding a layer, nodes in the hash table are sorted (in lexicographic ordering, as per Section 6.2) and written to disk. The previous layer is then deleted from disk.

For large datasets, it is possible that even one layer of the parent or order graph is too large to f t in RAM. We use a variant of *delayed duplicate detection* (DDD) [51] in our algorithm to utilize external memory when a layer will not f t in RAM. In DDD, search nodes are written to a f le on disk as soon as they are generated. After expanding a layer, an external-memory sorting algorithm, such as external-memory merge sort [37], is used to detect and remove duplicate nodes in the f le. The nodes in the f le are then expanded to generate the next layer of the search. In this manner, the search uses a minimal amount

of RAM; however, because all generated nodes are written to disk, much work is done reading and writing duplicates.

Rather than immediately writing all generated nodes to disk, we detect duplicates in RAM with a hash table. Figure 6.2(a), (b) and (c) show several examples of duplicate detection in RAM. Once the hash table reaches a user-def ned maximum size, its contents are sorted and written to a temporary f le on disk. The hash table is then cleared. At the end of each layer, the remaining contents of the hash table are sorted and merged with the temporary f les into a single sorted f le. An example of this operation is shown in Figure 6.2(d). Locality in our search allows us to detect many duplicates in RAM with the hash table and reduce external memory usage.



(a) The new node is not in the starting hash table; therefore, it is added to the hash table. (b) The new node is already in the hash table with a better score, so the new node is discarded. (c) The new node is already in the hash table, but the new node has a better score. The hash table is updated with the new node. (d) An example of delayed duplicate detection via external memory sorting. The input files are already sorted by key; however, there are some duplicates in multiple files. During the external memory merge, duplicates are removed and only the one with the best score is kept.

Figure 6.2

Examples of immediate and delayed duplicate detection.

6.5 Advantages of Frontier Breadth-First Branch and Bound

BFBnB enjoys many advantages over current state of the art methods. First, like A*, BFBnB can benefit from the pruning of Theorem 2 and the sparse parent graphs. Second, the layered structure we impose on the parent and order graphs ensures that we never need more than two layers of any of the graphs in memory, RAM or files on disk, at once. Third, because of the pruning described in Section 6.1, BFBnB does not waste resources expanding subnetworks which provably cannot result in an optimal structure. However, unlike A*, the pruning of BFBnB is dependent on the upper bound; a tight upper bound will result in more pruning. Finally, the delayed duplicate detection method we use lifts the requirement that a single layer fits in RAM. Because we do not resort to delayed duplicate detection until RAM is full, our algorithm takes advantage of all available RAM. By writing nodes to disk once RAM is full, we learn optimal Bayesian networks even when single layers of the search graph do not fit in RAM. The amount of available hard disk space and running time are the only limiting factors for the scalability of our algorithm.

None of the existing algorithms take advantage of the structure in the parent and order graphs when calculating $BestScore(X, \mathbf{V})$ or $Score(\mathbf{V})$. Singh and Moore [84] use a depth-first search ordering to generate the necessary scores and variable sets, while Silander and Myllymaki [82] use the lexicographic ordering over all of the variables. We use the lexicographic ordering only within each layer, not over all of the variables. The depth-first approach does not generate nodes in one layer at a time. The lexicographic ordering also does not generate all nodes in one layer at a time. Consider the first four nodes in the lexicographic order: $\{X_0\}, \{X_1\}, \{X_0, X_1\}, \{X_2\}$. Two nodes from layer 1 are generated,

then a node in layer 2; however, the next node generated is again in layer 1. Similarly, the seventh node generated is $\{X_0, X_1, X_2\}$ while the eighth node is $\{X_3\}$. Because generation of nodes from different layers is interleaved, these orderings require the entire graphs remain in memory (either in RAM or on disk). In contrast, our BFBnB algorithm generates nodes one layer at a time and thus needs at most two layers of the graphs in memory, plus the extra information to reconstruct the path. The delayed duplicate detection and solution reconstruction strategies allow us to store that information in external memory once RAM is full. Previous layers can safely be deleted.

6.6 Empirical Results

We empirically evaluated our BFBnB algorithm against DP and A* (from Chapter 5) for both space and time usage. We used both full and sparse parent graphs for BFBnB, but only show the results using the sparse parent graphs for A*. We compared the size of the full order graph, which a typical dynamic programming algorithm stores, to the maximum size of a layer in the order graphs that BFBnB has to store. Additionally, we compared the running times of all the algorithms.

Previous results found that memory is the main bottleneck restricting the size of learnable networks [70]. As our results in Figures 6.3 and 6.4 conf rm, algorithms which attempt to store entire parent or order graphs in RAM or on disk, such as DP and A* are limited to smaller sets of variables. BFBnB's duplicate detection strategy allows it to write partial search layers to hard disk when the layers are too large to f t in RAM, so it can learn optimal Bayesian network structures regardless of the amount of RAM. Consequently, hard

disk space and running time are its only limiting factors. The inexpensive cost of hard disks coupled with distributed f le systems can potentially erase the effect of memory on the scalability of the algorithm. The runtime results show that BFBnB not only takes much less space, but also runs several times faster than the DP algorithm.

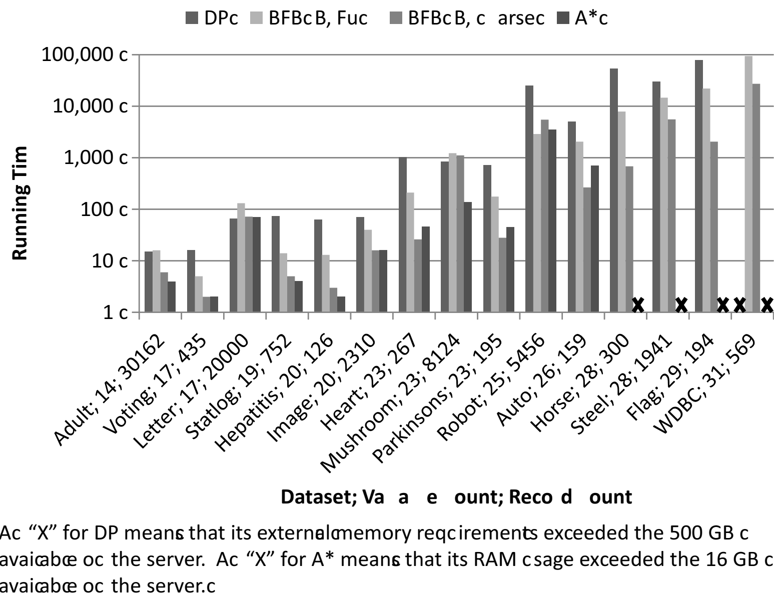


Figure 6.3

Runtime comparison of DP, BFBnB and A*.

On the largest dataset, $WDBC$ (31 variables and 569 records), we learned the optimal network in 93,682 seconds (about 26 hours) using full parent graphs. The time was reduced to 27,243 seconds (about 8 hours) using the sparse parent graphs. We also attempted to use DP, but its external memory usage exceeded the 500 gigabytes of hard disk space on the server. Figure 6.5 shows the total memory consumption of our algorithm on the largest layers of the $WDBC$ search using full parent graphs. Very little memory is used before

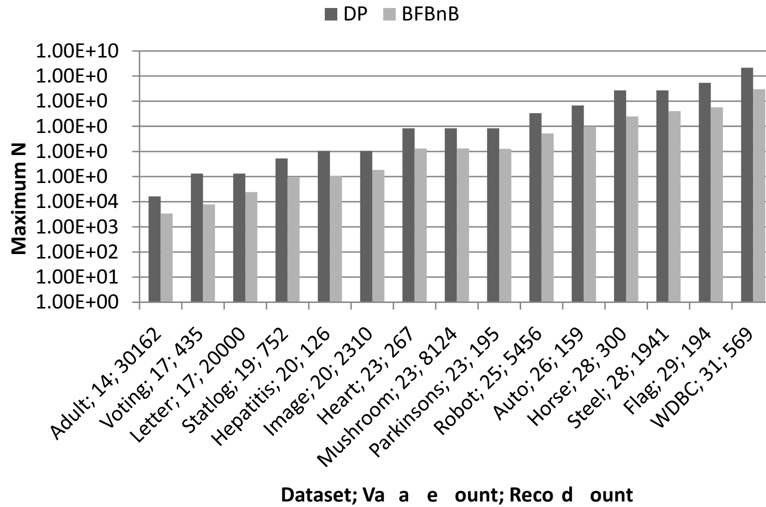


Figure 6.4

Comparison of order graph nodes stored in memory at once by DP and BFBnB.

layer 9, and after layer 23, the memory consumption does not change much because the layer sizes decrease exponentially. As the f gure shows, both of the middle layers use nearly 70 gigabytes of disk space. Most of this space is consumed by the parent graphs, so it is freed after each layer. Using sparse parent graphs eliminates all of that external memory usage. Assuming that the running time and size of the middle layers double for each additional variable, which is a rough pattern from Figures 6.3 and 6.4, our algorithm could learn a 36-variable network in about one month using approximately 2 terabytes of hard disk space and a single processor when using the full parent graphs. This suggests that our method should scale to larger networks better than the method of Parviainen and Koivisto [70]. They observe that their implementation would take 4 weeks on 100 processors to learn a 31-variable network, and, even with coding improvements and massive parallelization, only networks up to 34 variables would be possible.

As with A*, the sparse parent graphs typically improved the running time by over an order of magnitude compared to the full parent graph implementation. However, the sparse parent graphs did cause the algorithm to run slower on the *Sensor Readings* dataset. As the table shows, though, the "sparse" parent graphs were storing more than 900,000 scores. For this dataset, then they were not very sparse, and the bit operations took much longer because they were applied to so many scores during each iteration.

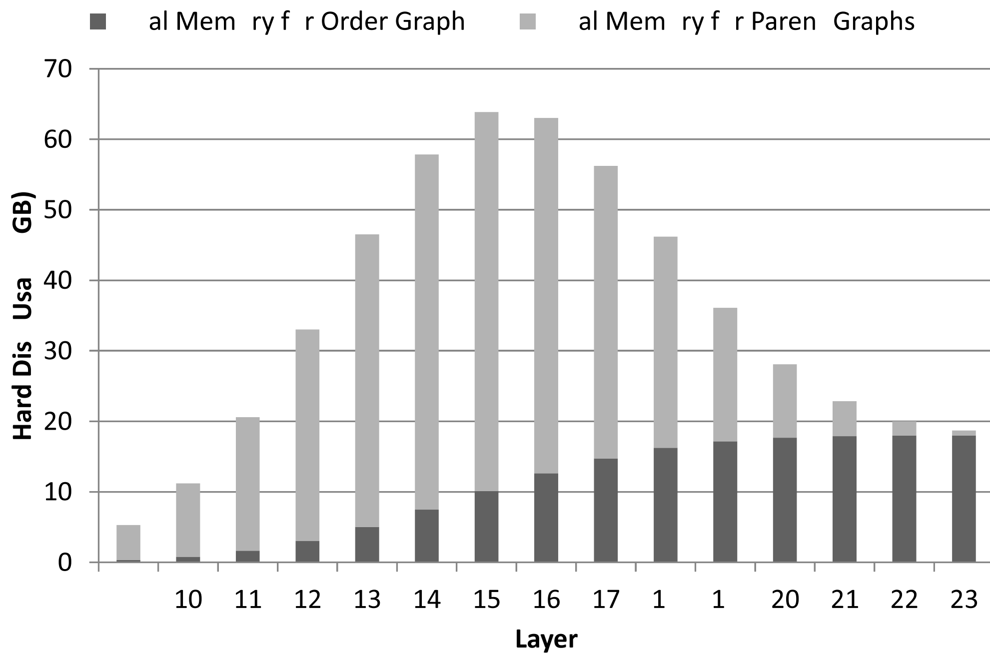


Figure 6.5

Hard disk usage for the *WDBC* dataset.

CHAPTER 7

ANYTIME DEPTH-FIRST BRANCH AND BOUND SEARCH

Our A* algorithm is shown to be an order of magnitude more eff cient than the dynamic programming algorithms. However, A* requires all the search information, including parent and order graphs, to be stored in RAM during the search, which makes the algorithm run out of memory for large datasets, even if using the parent graphs (see Chapter 8 for A* on very large datasets). BFBnB searches the order graph one layer at a time. By coordinating the parent and order graphs, most of the search information can be stored on disk and are only processed incrementally after being read back to RAM when necessary. The BFBnB algorithm was shown to be as eff cient as the A* algorithm but was able to scale to much larger datasets. Theoretically, the scalability of the BFBnB algorithm is only limited by the amount of disk space available.

However, the A* and BFBnB algorithms have a common limitation in that they do not f nd any solution until the very end of the search. If they run out of RAM or disk space before the search f nishes, they cannot provide any solution. As shown in Section 6.6, even if they do complete the search, a result my not be returned for hours or even days for large datasets. In many situations, we would desire the algorithms to exhibit *anytime* behavior; that is, we would like the algorithm to return a (potentially sub-optimal) solution quickly. Then, if time and other resources permit, the algorithm improves the solution

until converging upon the optimal network. If the search runs to completion, it guarantees to return the optimal network.

Several other exact algorithms do exhibit some form of anytime behavior. The branch and bound search of de Campos and Ji [21] has anytime behavior, but searches in the space of cyclic graphs. We showed in Section 5.5 that it is very slow to converge to the optimal network. Mathematical programming algorithms [44, 15] also have anytime behavior, but extra work is required to decode their intermediate results into a usable network structure. Additionally, MP was shown to be only comparable or slightly more eff cient the DP [44].

7.1 Anytime Algorithms

The notion of anytime search is not new. For example, Dean and Boddy [24, 4] present the notion in the context of planning under unknown time constraints. Standard DFS is a form of anytime search if the search is continued after the f rst solution is found. Unfortunately, graphs with many paths to each node, like the order graphs, can suffer an exponential increase in complexity using normal DFS (or normal DFBnB) because the same node can be expanded many times. Many algorithms have investigated approaches to minimize these node re-expansions. We f rst describe weighted A* because it serves as a basis for many of the algorithms, and then introduce three representative examples.

7.1.1 Weighted A*

Weighted A* (WA*) [74, 75, 71] is a variant of A* search in which the heuristic function is weighted by a factor ϵ. That is, $f(n) = g(n) + \epsilon \times h(n)$. By weighting the heuristic, it is no longer admissible. That is, the f value for a node may over-estimate the cost of

a path to the goal through this node. However, upon expanding a goal node, its cost is guaranteed to be no more than a factor of ϵ greater than the globally optimal solution [71]. For example, if $\epsilon = 1.05$, and we expand a goal node with cost f, then the globally optimal solution is guaranteed to be no more than 5% better than f.

7.1.2 Anytime Weighted A*

Anytime WA* [41] begins as a normal WA* algorithm; however, rather than stopping the search as soon as a solution is found, Anytime WA* continues to expand nodes. The score of the incumbent solution is then used to prune nodes based on their true, unweighted f cost, although nodes continue to be expanded based on the weighted value. As better paths to a goal are found, the incumbent solution is updated, which gives the algorithm its anytime behavior. Eventually, unless it is interrupted, the search expands or prunes all nodes in the search space and terminates with the optimal solution. Because of the weighted heuristic, Anytime WA* may f nd a better path to a closed node. In order to guarantee optimality of the f nal solution, Anytime WA* must re-expand those nodes.

7.1.3 Anytime Repairing A*

Anytime Repairing A*(ARA*) [58] adopts a similar strategy. It also starts as a normal WA* and runs until f nding a solution. Upon f nding the solution, the algorithm decreases ϵ and searches again. The solution is improved (or stays the same) at each iteration, so this algorithm also has anytime behavior. The process continues until it is interrupted or $\epsilon = 1$ and the algorithm terminates with an optimal solution. ARA* does not completely start from scratch for each search, though. Like AWA*, ARA* could also f nd a better

path to a node during the search. During each search iteration, the algorithm keeps a list of nodes closed during that iteration to which it finds a better path. Rather than immediately re-expanding those nodes, though, ARA* instead begins each iteration after the first with that list serving as the initial open list. The iterative process continues until better paths are found to no nodes.

7.1.4 Anytime Window A*

Anytime Window A* (AWA*) [1] adopts a slightly different approach to deliver anytime solutions. It is not based on WA*. Rather, it uses a type of sliding window to encourage deeper exploration of the search graph. Much like ARA*, the algorithm consists of a series of iterations. Instead of ϵ, AWA* uses a parameter w to control the size of the window. The algorithm keeps track of the depth of all nodes expanded during an iteration of the algorithm. After expanding a node in layer l, and nodes in layer $l - w$ are *frozen*. Nodes that are frozen are placed into a list to prevent them from being expanded. After the algorithm finds a solution on a particular iteration, the frozen nodes from the previous iteration become the new initial open list. This process continues until no nodes are frozen on an iteration.

7.2 Anytime DFBnB Network Learning Algorithm

There is no obvious way to convert the BFBnB algorithm into an anytime algorithm because the search expands one layer at a time in the order graph. The goal node is in the last layer and cannot be reached until the very end of the search. We can convert the A* search algorithm to an anytime algorithm by adopting a depth-first search strategy instead

of best-first search. Whenever the depth-first search reaches the goal node, a solution is found and can be used to update the best solution so far. Because, depth-first search requires retrieving $BestScore(\cdot)$ in a non-layered order, the full parent graphs are not practical for depth-first search. In this section, we present an *anytime depth-first branch and bound search algorithm* (DFBnB).

Table 7.1 gives pseudocode for our DFBnB algorithm. Unlike A* and BFBnB, we use the reverse order graph for this algorithm. Additionally, its design precludes full parent graphs. Section 7.2.1 describes an incremental update scheme which allows this algorithm to take more advantage of the sparse parent graphs than A* or BFBnB. A traditional shortcoming of DFBnB in graphs with many duplicates, like the order graph, concerns node re-expansions. We address this problem using a type of closed list described in Section 7.2.2. Because the start and goal nodes of the reverse order graph are different than those in the forward order graph, we use a heuristic described in Section 7.2.3. Unfortunately, because DFBnB does not expand nodes in best-first order, the closed list coupled with pruning causes some nodes to be inappropriately pruned. We use node re-expansions to ensure we consider the entire search space and guarantee optimality. In Section 7.2.4 we describe an iterative scheme to control node re-expansions while still guaranteeing optimality.

Reconstructing the optimal network at the end of the search uses the basic backtracking approach described in Section 5.3.

Table 7.1

A DFBnB search algorithm.

```
1: procedure EXPAND(U, valid, toRepair)
2:     if U = {} then
3:         h_exact(U) ← 0
4:     end if
5:     for each X ∈ U do
6:         BestScore(X, U \ {X}) ← scores_X[firstSetBit(valid_X)]
7:         g ← g(U) + BestScore(X, U \ {X})
8:         duplicate ← exists(g(U \ {X}))
9:         if g < g(U \ {X}) then g(U \ {X}) ← g
10:        if duplicate and g < g(U \ {X}) then toRepair ← toRepair ∪ {U, g}
11:        f ← h(U \ {X}) + g(U \ {X})
12:        if (!duplicate and f < optimal then
13:            for each Y ∈ U do
14:                valid'_Y ← valid_Y & ∼ parents_Y(X)
15:            end for
16:            expand(U \ {X}, valid')
17:        end if
18:        if h_exact(U) > BestScore(X, U \ {X}) + h_exact(U \ {X}) then
19:            h_exact(U) ← BestScore(X, U \ {X}) + h_exact(U \ {X})
20:        end if
21:    end for
22:    if optimal > h_exact(U) + g(U) then optimal ← h_exact(U) + g(U)
23: end procedure

24: procedure MAIN(D)
25:    for each X ∈ V do
26:        scores_X, parents_X ← calcScores(X, D); valid_X ← 1s
27:        for each Y ∈ V \ {X} do
28:            scores_X(Y) ← getScores(parents_X, Y)
29:        end for
30:    end for
31:    toRepair_l ← {V, 0}
32:    while |toRepair_l| > 0 do
33:        for each {V, g} ∈ toRepair_l do
34:            if g(V) > g then
35:                g(V) ← g
36:                expand(V, valid, toRepair_{l+1})
37:            end if
38:        end for
39:        toRepair_l ← toRepair_{l+1}
40:    end while
41: end procedure
```

7.2.1 Incremental Sparse Updates

In addition to its anytime behavior, DFBnB has another useful property. Because it completely searches one branch of the graph before jumping to another, it allows us to exploit another regularity in the order graph. A successor of a node in the reverse order graph removes exactly one variable (used as the leaf) from its predecessor. In Section 4.2.2, we described an eff cient technique to remove one parent from consideration at a time when using the sparse parent graphs. Therefore, as shown in Table 7.1, we can incrementally modify the $valid$ bit vectors within the main algorithm rather than using the $calcualteBestScore$ procedure from Table 4.5.

At each node $\mathbf{U}$, we make one variable X as a leaf (line 5) and select its optimal parents from among $\mathbf{U}$ (lines 6 - 8). We then check if that is the best path to the subnetwork $\mathbf{U}\backslash\{X\}$ (lines 10 - 12). Because X is no longer a valid parent, no decendents of $\mathbf{U}$ along this path can use X as a parent. We remove X as consideration as a parent by performing the bit operation $valid_Y \& \sim parents_Y^X$ for the other $Y \in \mathbf{U}$ (lines 15 - 17). We then recursively select optimal parents for the remaining variables (line 18). After backtracking to $\mathbf{U}$, we select another Y in $\mathbf{U}$ to use as a leaf. Because we did not modify $valid$, the call stack maintains the valid parents before removing X; we can easily perform the bit operations for Y and continue the search. Because we have no more than n bit vectors and the reverse order graph always has n layers, we store at most $O(n^2)$ bit vectors in memory at once.

7.2.2 Closed List and Backups

A potential problem with DFS in graphs with many paths to each node is generating duplicates. A traditional DFS algorithm does not perform duplicate detection; therefore, much work can be wasted in re-expanding duplicate nodes. Our search graph contains many duplicates; a node in layer l is generated l times. In order to combat this problem, our algorithm uses a hash table to detect duplicate nodes (lines 9, 14). However, because of the depth-first search strategy, we are not guaranteed that we have the optimal path to a node the first time we expand it. On the other hand, because we always consider all of a node's descendants before backtracking to it, we know the exact distance between that node and the goal, $h_{exact}(\mathbf{U})$ before it is generated again. To take $h_{exact}(\mathbf{U})$ as correct, though, we must assume that none of its descendents are inadmissibly pruned. We discuss this issue in more detail in Section 7.2.4. When backtracking, we can compute $h_{exact}(\mathbf{U})$ by calculating for each successor $\mathbf{R}$ the total distance between $\mathbf{U}$ and $\mathbf{R}$ and between $\mathbf{R}$ and the goal, and finding the minimum distance among them (lines 20 - 22). Trivially, the distance of an immediate predecessor of the goal is just the distance between it and the goal. We pass this information up the call stack to calculate the distances for predecessor nodes. Then, the next time $\mathbf{U}$ is generated, we sum the distance on the current path and $h_{exact}(\mathbf{U})$. If it is better than the existing best path, $optimal$, then we update the best path found so far (lines 24 - 26). We store all values of $h_{exact}(\mathbf{U})$ in the hash table.

7.2.3 Heuristic Function for the Reverse Order Graph

The eff ciency of the depth-f rst search can be signif cantly improved by using a lower bound for pruning. The best solution found so far is a trivial upper bound for the optimal solution. If we can also estimate a lower bound for all the paths that pass through the current search node, and the lower bound is already worse than the upper bound solution, the current node can be immediately pruned as it will not lead to any better solution. Since the new order graph has a different goal node from the original graph, we cannot use the heuristic function in Equation 5.1.

At any point in the search, we have a set of variables remaining which must form the rest of the network. We know the scores which could possibly be used for all the remaining variables. By consulting the bit vectors $valid$ at a particular node $\mathbf{U}$, we can identify the best scores those variables could possibly have along the path which includes $\mathbf{U}$; that is, for all X in $\mathbf{U}$, we can calculate $BestScore(X, \mathbf{U})$. By summing over these scores, we can calculate a lower bound on the optimal subnetwork over $\mathbf{U}$, or the distance from $\mathbf{U}$ to the goal node, i.e., we use the following new heuristic function h^*.

Def nition 2

$$h^*(\mathbf{U}) = \sum_{X \in \mathbf{U}} BestScore(X, \mathbf{U} \setminus \{X\}). \tag{7.1}$$

The heuristic is admissible because it allows the remaining variables to select their optimal parents from among all of the other remaining variables. This has the effect of relaxing the acyclic constraint on those variables. The following theorem proves the heuristic is also consistent.

Theorem 4 *h^* is consistent.*

Proof: For any successor node $\mathbf{R}$ of $\mathbf{U}$, let $Y \in \mathbf{U} \setminus \mathbf{R}$. We have

$$\begin{aligned} h^*(\mathbf{U}) &= \sum_{X \in \mathbf{U}} BestScore(X, \mathbf{U} \setminus \{X\}) \\ &\leq \sum_{X \in \mathbf{R}} BestScore(X, \mathbf{R} \setminus \{X\}) + BestScore(Y, \mathbf{U} \setminus \{Y\}) \\ &= h^*(\mathbf{R}) + c(\mathbf{U}, \mathbf{R}). \end{aligned}$$

The inequality holds because the variables in $\mathbf{R}$ have fewer parents to choose from after making Y a leaf. Hence, h^* is consistent. □

Because we do not expand nodes in a best-f rst order, the consistent heuristic does not allow us to discard duplicate nodes; however, we can use the heuristic to prune parts of the search space which cannot possibly be on the optimal path from the start to the goal node (lines 13 - 14). Computing the heuristic for any node $\mathbf{U}$ is linear in the number of variables remaining in $\mathbf{U}$.

7.2.4 Repairing Inconsistent Nodes

Integrating a closed list to maintain h_{exact} within the DFBnB framework greatly minimizes the number of node re-expansions that must occur; however, it also causes subtle problems when pruning. As mentioned several times, the f rst time we expand a node n it may not have its optimal g-cost. We expand n with the discovered g value, $g'(n)$. The g cost of its successors, $g(s) = g'(n) + c(n, s)$, then include the over-estimate present in $g'(n)$. If the difference between the true $g(n)$ (i.e., the shortest path from the start node to n) and $g'(n)$ is Δ_g and the difference between the f cost of the current incumbent and

$f'(s) = g'(n) + cost(n, s) + h(n)$ is less than Δ_g, then s will be inadmissibly pruned. Furthermore, it could happen that all of s's predecessors are initially expanded with sub-optimal g costs and s is always inadmissibly pruned. Because all of s's predecessors are in the closed list, though, they will not be re-expanded. Then, even though s could have been a part of the optimal solution, it was never expanded because of inadmissible pruning.

This is very similar to the problem faced by Anytime WA*, ARA*, AWA* and many other best-first algorithms which use inadmissible heuristics. Consequently, we adopt a similar solution. In constrast to those algorithms, we use a consistent heuristic function; however, as described in the example, we can expand a node that has a sub-optimal g-cost. During each iteration, we keep a list, similar to the inconsistent list of ARA*, that tracks nodes to which we find a better path. We do not re-expand nodes immediately. Instead, we add them to the list and note the new g cost. Once the current iteration of search finishes, we repair the nodes in the list by expanding them with the new g cost. We repeat this iterative process until no nodes are added to the list during an iteration.

We prefer the DFS strategy to a weighted best-first strategy here because it does not incur the overhead associated with maintaining a priority queue. Additionally, the DFS is guaranteed to find the first solution on the $n + 1^{th}$ expansion, while the WA*-based algorithms may take longer to generate the first solution. Furthermore, the DFS strategy allows us to more efficiently perform the incremental bitwise operations to the sparse parent graph, while best-first strategies require we start from scratch for each calculation.

7.2.5 Advantages of DFBnB

The DFBnB algorithm offers several advantages compared to A*, BFBnB and other structure learning algorithms. Eventually DFBnB will either converge to the optimal solution, or output the best solution found so far whenever it runs out of time or memory or has to be stopped early. A* cannot complete if it runs out of RAM. Although BFBnB is not restricted by RAM, it is limited by the amount of available hard disk space. On large datasets, BFBnB can easily require terabytes of hard disk; if this amount is not available, then BFBnB does not return any network. In contrast, even if DFBnB runs out of resources before provably f nding the optimal network, it can still output the best network found.

As we show in Section 7.3, the algorithm has very good anytime behavior compared to the branch and bound algorithm of de Campos and Ji [19] and even Optimal Reinsertion [64], a local search algorithm. In fact, for many datasets, much of the search time is spent simply proving the optimality of the solution. Thus, in paractice, the algorithm can often be stopped very early and still give the optimal solution.

7.3 Empirical Results

We tested the DFBnB algorithm against several state of the art structure learning algorithms.

7.3.1 Comparison of Anytime Behavior

First, we compared the anytime behavior of DFBnB to that of BB and OR on four datasets of up to 57 variables: $Auto$, $Flag$, $Water$ and $Lung$. We chose to compare to BB

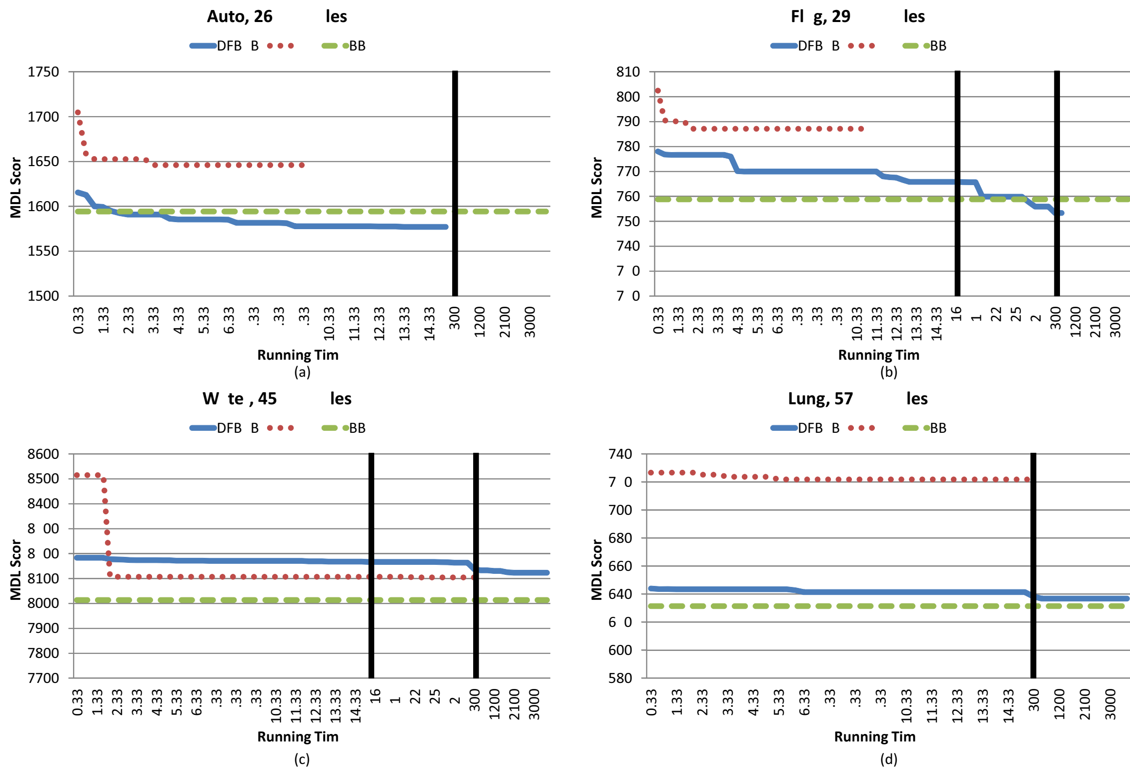
because of its anytime behavior. We compare to OR as a representative for local search techniques because several studies [89, 91] suggest that it performs well on a wide variety of datasets. OR has several tunable parameters, including scoring function and maximum parent count. The MDL complexity penalty term implemented in DFBnB and BB differs from that of OR. (Implementations of other local search algorithms, such as sparse candidate [36], also varied in calculated scores, even for the same network structure.) To account for this difference, the structures learned by OR were rescored with the scores used by DFBnB and BB which always assigned the same scores to equivalent structures. For a dataset of size N and the MDL scoring function, no variable can have more than $k = \log \frac{2N}{\log N}$ parents in the optimal network [90]; we used that value of k as the maximum number of parents for each dataset. We ran OR with the given parameters. We then plotted the scores of the networks learned by each algorithm as a function of time. To perform these experiments, we allowed the algorithms to run up to one hour (3600s). The runtimes for DFBnB and BB include only the time spent on search; they do not include times to calculate the local scores.

The convergence curves of these algorithms on the datasets are shown in Figures 7.1. In these experiments, OR was always the f rst algorithm to terminate. OR terminated because it reached a local optimum and was unable to escape. Only once on the water treatment dataset did it f nd a slightly better solution than DFBnB. BB did not f nish searching within the time limit for any of the datasets. Also, the convergence curves of BB stayed f at for all the datasets. That means BB was not able to improve any of the initial solutions found by the greedy algorithm it uses to initialize its bound. The true anytime behavior of BB is

thus unclear from the results. In comparison, DFBnB finds all the solutions by itself, so its curves provide a reliable indication of its anytime behavior. On all the datasets, DFBnB continuously finds better solutions during the search, and was also able to find and prove the optimality of its solutions on one of the datasets. On two occasions it was able to find solutions better than the initial solutions of BB. DFBnB could also benefit from the better initial solutions found by the greedy algorithm used in BB.

7.3.2 Comparison of Running Time

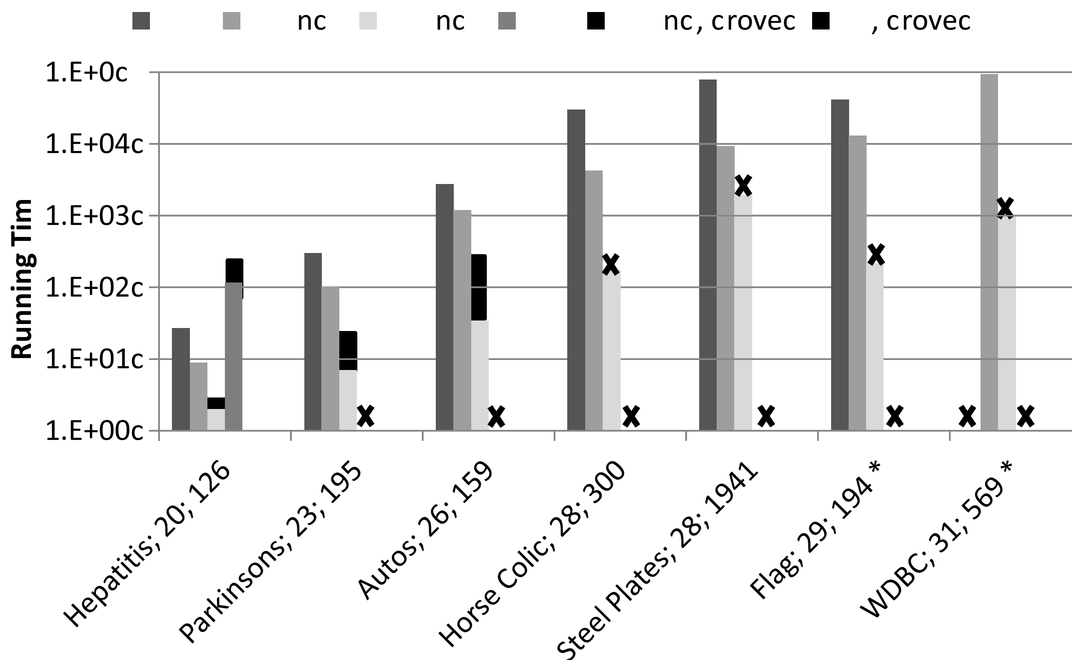
Finally, we compared the running time of DFBnB to those of DP, BFBnB and BB. DFBnB and BB were again given a one hour limit. We let BFBnB and DP run longer in order to obtain the optimal solutions for evaluation. For this comparison, we considered both the time to *find* the best structure and to *prove* its optimality by DFBnB and BB. The results in Figure 7.2 demonstrate that DFBnB finds the optimal solution nearly two orders of magnitude faster than current state of the art algorithms. Furthermore, DFBnB proved the optimality nearly an order of magnitude more quickly when it had enough RAM. However, DFBnB does not take advantage of disk. The program stops when its hash table fills RAM, so it is unable to prove the optimality for some of the searches. Nevertheless, the search finds optimal solutions for all but two of the datasets. We verified this using the results from BFBnB. BB proved the optimality of its solution on only the smallest dataset. On all other cases, it did not improve its initial solutions. These echo the results of Figure 7.1. BB does not improve its initial solution quickly. DFBnB found better solutions than BB on all these datasets.



AlgorithB rB givBn up to onB hour (3600Bs) of runtiB .BA blBck bBr indicBtB n incrB in thB tiB cBB.B ThB runtiB do not includB thB tiB tBkBn to cBlculBtB scorB bBcBuB thBt iB thB ong BllB lgorithB .B InB ll cB ,B not only did not find thB optiB l nBtwork, but did not iB provB upon thB olution foundBby itB initiBl grB dy B rch.B(a)BOR tBrminBtBdB ftBr rB chingB locBl optiB uB t 10B condB.B FBnB tBrmBnBtBdB ftBr B findingB nd proving thB optiB l solutionB ft r 15B condB. Th optiB l corB i 1577.16. (b) OR tBrminBt dB ft r rB chingB locBl optiBuB t 11B condB. F n t rminBt d ft r filling RAMB t 415B condB. Th optiB lB scorB i 2748.05. (c) OR tBrminBt d ft r rB ching B locBl optiB uB t 40B condB. (d) OR tBrminBt d ft r B rB chingB locBl optiB uB t 45B condB.B

Figure 7.1

Anytime comparison of DFBnB, OR and BB.



Dataset; Va a e ount; Reco d ount

and c nc were given an arbitrary amount of time to complete because of their c efficient use of external memory. c nc and c were given a maximum running time c of one hour. cor c , an "X" for a dataset means the external memory usage exceeded c 00 GBccor c nc and c , an "X" means it did not prove the optimality of the solution c found. An "X" at time 0 for c means that c did not improve its initial greedy solution c and did not find the optimal solution. c nc found the optimal solution, though may c not have proved it, for all datasets which do not have an "*" beside the name.c

Figure 7.2

Comparison on the runtimes of DP, BFBnB, DFBnB and BB to f nd optimal networks.

CHAPTER 8

THE K-CYCLE CONFLICT HEURISTIC

All of the proposed algorithms use an admissible heuristic to safely ignore parts of the search space. A* uses Equation 5.1 to leave parts of the search space unexpanded, while BFBnB uses the same heuristic function to prune away unpromising parts of the search. Similarly, DFBnB utilizes Equation 7.1 to prune the reverse order graph. Both of these heuristic functions relax the acyclic constraint of Bayesian networks so that each remaining variable can freely choose optimal parents from other variables. The heuristic provides an *optimistic* estimation of how good a solution can be and is admissible. This simiple relaxation does not consider interactions among the selected parents, though. Therefore, it may introduce many directed cycles into the relaxed problem. If a graph has many cycles, the bound may be quite loose and limit the effectiveness of pruning.

In this chapter, we propose a tighter admissible heuristic which considers and eliminates directed cycles within small groups of remaining variables. The resulting technique, called the *k-cycle conf ict heuristic*, is a type of additive pattern database [30]. *Pattern databases* [14] calculate an admissible heuristic value for a problem by solving a relaxed version of the problem optimally. The cost of the exact solution of the relaxed problem is admissible for the original problem [71]. In general, multiple problems in the original state space are relaxed to the same problem. Therefore, the relaxed problems form an abstract

state space in which multiple original states map to the same relaxed state. The relaxed state is also called a *pattern*. A pattern database consists of exact costs for the patterns, which can be looked up as an admissible heuristic for the original states. For Bayesian network structure learning, a pattern consists of a set of variables. We can then create multiple pattern databases by relaxing it in different ways. If a set of relaxed problems have no interactions between them, the costs of the pattern databases can be added together to obtain an admissible heuristic, which is why the method is called an *additive* pattern database. Otherwise, the only way to obtain an admissible heuristic is to take the maximum cost of the pattern databases.

We consider two version of the k-cycle conflict heuristic. The first version dynamically splits the remaining variables into small groups in an attempt to maximize the heuristic value. In the second version, we statically split the variables into groups at the beginning of the search and only break cycles among variables in the same group. Both versions of the heuristic are adapted to A* and BFBnB.

8.1 A Motivating Example

According to Equation 5.1, the heuristic estimate of the start node in the order graph allows each variable to choose optimal parents from all the other variables. Suppose the optimal parents for X_1, X_2, X_3, X_4 are $\{X_2, X_3, X_4\}$, $\{X_1, X_4\}$, $\{X_2\}$, $\{X_2, X_3\}$ respectively. The parent sets selected by the heuristic are shown as the directed graph in Figure 8.1. Since the acyclic constraint is ignored, directed cycles are introduced, e.g., between X_1 and X_2. However, we know the final solution cannot have cycles. Three sce-

narios are possible between X_1 and X_2 in the optimal Bayesian network: (1) X_2 is a parent of X_1 (so X_1 cannot be a parent of X_2), (2) X_1 is a parent of X_2, or (3) neither of them is a parent of the other. Therefore, we can break the cycle to achieve a tighter bound. Before discussing how to do that, we f rst introduce the following theorem.

Theorem 5 *Let* $\mathbf{U}$ *and* $\mathbf{V}$ *be two candidate parent sets for* X*, and* $\mathbf{U} \subset \mathbf{V}$*, then* $BestScore(X, \mathbf{V}) \leq BestScore(X, \mathbf{U})$.

The theorem has appeared in many earlier papers, e.g. [54], and simply means that a better score can be obtained if a larger set of parent candidates is available to choose from. Due to the theorem, the third case outlined earlier is guaranteed to be worse than the other two because one of the variables has fewer parents to choose from. Between the f rst two cases it is unclear which one provides a better value, so we take the minimum of them. Consider the f rst case: We have to delete the arc $X_1 \rightarrow X_2$ to rule out X_1 as a parent of X_2. After that we have to let X_2 to reselect optimal parents from $\{X_3, X_4\}$. The deletion of the arc alone cannot produce the new bound; we must check the second best, third best, etc., parent sets for X_2 until we f nd one that does not include X_1. To f nd the total bound for X_1 and X_2, we sum together the original bound for X_1 and the new bound for X_2. We call that b_1. The second case is also handled similarly; we call that bound b_2. Because the total bound for X_1 and X_2 must be optimistic, we take the minimum of b_1 and b_2. The new heuristic is clearly still admissible, because we still allow cycles among other variables.

Often, the simple heuristic introduces multiple cycles. The graph in Figure 8.1 has a cycle between X_1 and X_2. It also has a cycle between X_2 and X_4. Because both cycles

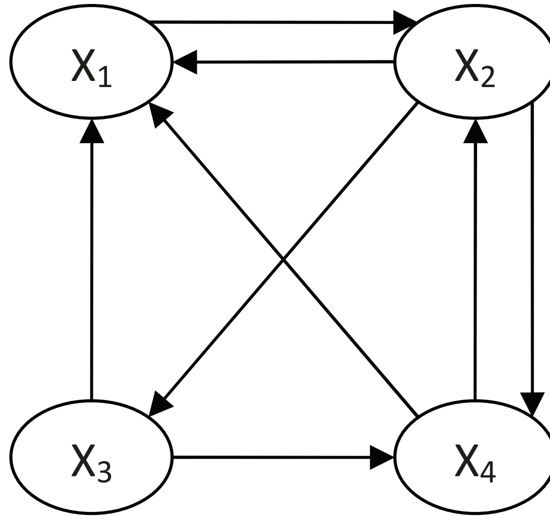


Figure 8.1

A directed graph representing the heuristic estimate for the start search node.

include X_2, we say they *overlap*. Overlapping cycles cannot be broken independently. For example, suppose we break the cycle between X_1 and X_2 by setting the parents of X_2 to be $\{X_3\}$. Then we also break the cycle between X_2 and X_4, but introduce a new cycle between X_2 and X_3. As described in more detail below, we group variables together and do not break cycles between variables in different groups. So, if X_2 and X_3 were in different groups, we would not break that cycle.

8.2 Dynamic k-cycle Conflict Heuristic

The dynamic k-cycle conflict heuristic can be calculated by using sparse parent graphs to perform a breadth-first search through the first k layers in the reverse order graph. The $createDynamicPD$ procedure of Table 8.1 gives pseudocode for constructing the pattern database. A node $\mathbf{U}$ in the reverse order graph represents a subnetwork over the variables

$\mathbf{V} \setminus \mathbf{U}$ in which each variable X selects its optimal parents from among all of the variables which are removed after X. We call the sum of these scores as $Score'\mathbf{U})$. This score is also gives a lower bound for any subnetwork in the forward order graph that does not include $\mathbf{U}$. Therefore, the cost for the pattern $\mathbf{V} \setminus \mathbf{U}$ is equal to $Score'(\mathbf{U})$. We can also evaluate the quality of a pattern by comparing the *difference* between $Score'(\mathbf{U})$ and $h(\mathbf{U})$, which we call $\delta_h(\mathbf{U})$. A difference of 0 indicates that there is no benef t to using the pattern, so the optimal parent selections for the pattern variables do not include any cycles. A large difference suggests that the optimal parent selections include cycles, and breaking those cycles improves the heuristic. After calculating $\delta_h(\mathbf{U})$ for all subsets of variables up to size k, we prune all patterns which do not have a higher $\delta_h(\mathbf{U})$ than any of its subsets. The pruning can signif cantly reduce the size of the pattern database, which is important when using the dynamic pattern database during the heuristic search. Finally, we order the patterns in order of decreasing δ_h. That is, patterns that offer the most improvement over the simple heuristic are f rst in the list.

Once the dynamic k-cycle conf ict heuristic is computed, we can use it to calculate the heuristic value for any node during any of the search algorithms we have presented. For a node $\mathbf{U}$ in A* or BFBnB, which use the forward order graph, we partition the remaining $\mathbf{V} \setminus \mathbf{U}$ variables into a set of non-overlapping patterns. Because the patterns do not interact, we then sum together their cost to f nd the total heuristic value of the node. The algorithms require no other modif cations to incorporate the tighter bounds offered by the pattern database.

Ideally, we would like to find the partition with the highest total cost, which corresponds to the tightest heuristic value. We can then find the optimal partition by solving the *maximum weighted matching problem* on the graph [30]. For $k = 2$, we can define a matching graph in which vertices represent variables and edges between variables have a weight equal to the cost of the pattern which comprises those two variables. In this problem, we select a set of edges from the graph so that no two edges share a vertex and the total weight of the edges is maximized. The edges correspond to the patterns we should select. The matching problem can be solved in $O(n^3)$ time [69], where n is the number of vertices.

Unfortunately, for $k > 2$, the matching graph contains *hyperedges* the connect up to k vertices to represent the larger patterns. For example, a pattern for three variables would induce a hyperedge connecting three vertices. We must again select edges (for patterns of size 2) and hyperedges (for larger patterns) that maximize the total weight. The higher-order maximum weighted matching problems are NP-hard [38]. Therefore, calculating the heuristic value optimally would require solving an NP-hard problem for each search node.

We use a greedy algorithm given in the $h_{dynamic}$ procedure of Table 8.1 to calculate the heuristic value to keep the computation efficient. Assume we must partition $\mathbf{U}$ into non-overlapping patterns. Because the patterns are sorted by δ_h, we select the first pattern $\mathbf{P}$ which is a subset of $\mathbf{U}$. We then search for the next pattern which is a subset of $\mathbf{U} \setminus \mathbf{P}$. We repeat this process until removing all variables. The total cost of the selected patterns is returned as the heuristic value. This method is an example of a *dynamically partitioned pattern databases* [30] because the patterns are partitioned dynamically for each node in

an attempt to find the tightest possible bound. We refer to this heuristic as the *dynamic pattern database* for short.

8.3 Static k-cycle Conflict Heuristic

Computing the heuristic value for a search node using the dynamic pattern database even with the greedy method is much more expensive than the simple heuristic in Equation 5.1 because the list of patterns is scanned for each node. Consequently, each node expansion takes more time, so the total running time can be longer even though the tighter heuristic results in more pruning.

We also designed a *statically partitioned pattern database* [30] based on the k-cycle conflict heuristic. In this approach, we statically divide all variables into a set of disjoint groups at the beginning of the search. Then, we create a pattern database for each group using the $createStaticPD$ procedure from Table 8.2. To construct the pattern database for a static group $\mathbf{V_i}$, we again use sparse parent graphs to perform a breadth-first search in the reverse order graph; however, we only consider edges in the reverse order graph which correspond to selecting elements of $\mathbf{V_i}$ as leaves. In essence, this allows variables in $\mathbf{V_i}$ to always use $X \in \mathbf{V} \setminus \mathbf{V_i}$ as candidate parents but detects and eliminates cycles among variables in $\mathbf{V_i}$.

Consider a problem with variables $\{X_1, ..., X_8\}$. We simply divide the variables into two equal-size groups, $\{X_1, ..., X_4\}$ and $\{X_5, ..., X_8\}$. For each group, say $\{X_1, ..., X_4\}$, we create a pattern database that contains the costs of *all* subsets of $\{X_1, ..., X_4\}$ and similarly for $\{X_5, ..., X_8\}$.

Table 8.1

Dynamic k-cycle conf ict heuristic.

1: **procedure** CREATEDYNAMICPD(k)
2: $\quad PD_0(\mathbf{V}) \leftarrow 0$
3: $\quad \delta_h(\mathbf{V}) \leftarrow 0$
4: $\quad$ **for** $l = 1 \rightarrow k$ **do**
5: $\quad\quad$ **for** each $\mathbf{U} \in PD_{l-1}$ **do**
6: $\quad\quad\quad expand(\mathbf{U}, l)$
7: $\quad\quad\quad checkSave(\mathbf{U})$
8: $\quad\quad\quad PD(\mathbf{V} \setminus \mathbf{U}) \leftarrow PD_{l-1}(\mathbf{U})$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: $\quad$ **for** each $X \in PD \setminus save$ **do**
12: $\quad\quad$ delete $PD(X)$
13: $\quad$ **end for**
14: $\quad$ sort($PD : \delta_h$)
15: **end procedure**

16: **procedure** EXPAND($\mathbf{U}, l$)
17: $\quad$ **for** each $X \in \mathbf{U}$ **do**
18: $\quad\quad g \leftarrow PD_{l-1}(\mathbf{U}) + BestScore(X, \mathbf{U} \setminus \{X\})$
19: $\quad\quad$ **if** $g < PD_l(\mathbf{U} \setminus \{X\})$ **then** $PD_l(\mathbf{U} \setminus \{X\}) \leftarrow g$
20: $\quad$ **end for**
21: **end procedure**

22: **procedure** CHECKSAVE($\mathbf{U}$)
23: $\quad \delta_h(\mathbf{U}) \leftarrow g - \sum_{Y \in \mathbf{V} \setminus \mathbf{U}} BestScore(Y, \mathbf{V} \setminus \{Y\})$
24: $\quad$ **for** each $X \in \mathbf{V} \setminus \mathbf{U}$ **do**
25: $\quad\quad$ **if** $\delta_h(\mathbf{U}) > \delta_h(\mathbf{U} \cup \{X\})$ **then** $save(\mathbf{U})$
26: $\quad$ **end for**
27: **end procedure**

28: **procedure** $h_{dynamic}(\mathbf{U}, X)$
29: $\quad h \leftarrow 0$
30: $\quad \mathbf{R} \leftarrow \mathbf{U}$
31: $\quad$ **for** each $\mathbf{S} \in PD$ **do**
32: $\quad\quad$ **if** $\mathbf{S} \in \mathbf{R}$ **then**
33: $\quad\quad\quad \mathbf{R} \leftarrow \mathbf{R} \setminus \mathbf{S}$
34: $\quad\quad\quad h \leftarrow h + PD(\mathbf{S})$
35: $\quad\quad$ **end if**
36: $\quad$ **end for**
37: $\quad$ **return** h
38: **end procedure**

We store each pattern database as a hash table. Typically, the pattern databases are much smaller than the size of the order graph, so there is no need to order or prune the patterns. For example, the order graph for a 30 variable dataset has roughly 1 billion nodes (2^{30}). If we create 3 pattern database of size 10 each, in total, they would only comprise about 3 thousand nodes (3×2^{10}). We refer to this heuristic as the *static pattern database* for short.

Using the static pattern databases is simpler than the dynamic pattern databases, as shown in the h_{static} procedure of Table 8.2. For the forward order graph used in A* and BFBnB, we partition the variables which have not yet been added as leaves (i.e., $\mathbf{V} \setminus \mathbf{U}$) according to the static grouping. We then look up the cost of the patterns in the appropriate pattern databases and sum them together. Since each node expansion affects only a single node expansion, we can incrementally compute the heuristic value. As with dynamic pattern databases, the algorithms require no other modif cation to incorporate the static pattern databases.

8.4 Advantages of the k-cycle Conf ict Heuristic

Both version of the k-cycle conf ict heuristic offer obvious advantages to all of three of the described algorithms. As described in Sections 8.2 and 8.3, incorporating them into the search algorithms requires little additional effort, in terms of code complexity, additional runtime and memory overhead. As we show in Section 8.5, their tighter bound reduces the number of nodes expanded by A* and increases the number of nodes pruned by BFBnB. These result in improved runtimes and memory usage for all of the algorithms.

Table 8.2

Static k-cycle conf ict heuristic.

```
1:  procedure CREATESTATICPD(V^i)
2:      PD^i_0(V_i) ← 0
3:      for l = 1 → |V| do
4:          for each U ∈ PD^i_{l−1} do
5:              expand(U, l)
6:              PD^i(V_i \ U) ← PD^i_{l−1}(U)
7:          end for
8:      end for
9:  end procedure

10: procedure EXPAND(U, l)
11:     for each X ∈ U do
12:         g ← PD^i_{l−1}(U) + BestScore(X, U \ {X} ⋃_{j≠i} V_j)
13:         if g < PD^i_l(U \ {X}) then PD^i_l(U \ {X}) ← g
14:     end for
15: end procedure

16: procedure h_static(U, X)
17:     h ← 0
18:     for each V_i ∈ V do
19:         h ← h + PD^i(U ∩ V_i)
20:     end for
21:     return h
22: end procedure
```

8.5 Empirical Results

We tested the k-cycle conf ict heuristic on the A* and BFBnB algorithms by comparing to the heuristics given in Equation 5.1. In all cases, we used sparse parent graphs. The experiments were performed on a PC with 3.07 GHz Intel i7 processor, 16 GB of RAM, 500 GB of hard disk space, and running Ubuntu 10.10. We used benchmark datasets from the UCI machine learning repository [33] to test the algorithms. For all the datasets, records with missing values were removed. All variables were discretized into two states around means.

8.5.1 Improvement from the Pattern Database Heuristics

The k-cycle conf ict heuristic has two versions: dynamic and static; each of them can be parameterized in different ways. We applied various combinations of the new techniques to A* and BFBnB on the datasets Autos and Flag. For the dynamic pattern database, we varied k from 2 to 4. Empirically, the performance of larger values of k deteriorated quickly (results not shown). For the static pattern databases, we tried groupings 9-9-8 and 13-13 for the Autos dataset and groupings 10-10-9 and 15-14 for the Flag dataset. We selected these groupings because they result in roughly equally-sized pattern databases for each grouping. Felner *et al.* [30] used a similar grouping scheme for computing a static pattern database for the sliding tile puzzle. The results are shown in Table 8.3.

Both the static and dynamic pattern databases helped both algorithms improve their eff ciency and scalability. A* with both the simple heuristic and the static pattern database with grouping $10 - 10 - 9$ ran out of memory on the Flag dataset. The other pattern

database heuristics enabled A* to finish successfully. The dynamic pattern database with $k = 2$ significantly reduced the number of nodes expanded for all algorithms, and $k = 3$ usually granted further improvement. Further increasing k to 4 was not as beneficial, though; often the runtime increased, and sometimes more nodes were expanded. The longer running time, even when the total number of nodes expanded is reduced, results because of the larger size of the pattern database. Our greedy scanning method to calculate the heuristic is linear in the size of the pattern database. Therefore, larger databases increase the time required to compute the heuristic. That inefficiency gradually outweighed the benefit brought by the tighter heuristic. The greedy scanning technique also explains the occasional increase in expanded nodes from $k = 3$ to $k = 4$. Given an optimal partitioning of the remaining variables, we believe that larger k always results in a better (or at least the same) heuristic. However, the greedy partitioning may leave many variables nearly unconstrained. For example, suppose the remainig variables for a node in the search are $\{X_0, X_1, X_2, X_3, X_4\}$ and that the optimal partition is $\{X_0, X_2, X_4\}$, $\{X_1, X_3\}$. If $\delta_h(\{X_2, X_3\}) > \delta_h(\{X_0, X_2, X_4\})$, though, the greedy partitioning could result in $\{X_2, X_3\}, \{X_0\}, \{X_1\}, \{X_4\}$. That is, X_0, X_1 and X_4 are unrestricted in their choice of parents. Based on these results, we concluded that $k = 3$ is the best parametrization for the dynamic pattern database.

For the static pattern databases, we were able to test much larger groups because we do not enumerate all subsets up to size k like the dynamic pattern database does. Rather, we enumerate the subsets of each grouping of variables. The results suggest that larger

groupings tend to result in tighter heuristic values because fewer nodes were expanded when using the larger groupings.

The sizes of the static pattern databases are typically much larger than the dynamic pattern databases. However, they are still quite small in comparison to the number of expanded nodes in all cases, so it is cost effective to try to compute larger pattern databases to achieve better search efficiency. The results show that the best static pattern databases typically helped all three algorithms to achieve better time efficiency than the best dynamic pattern database. Sometimes the better time efficiency is achieved when the number of expanded nodes is larger for the static pattern databases. Again, the reason is calculating the heuristic value for a node is more efficient in the static pattern databases. Therefore, the selection between static and dynamic pattern databases embodies a space-time tradeoff. These results mirror those for using additive static and dynamic pattern databases for the sliding tile puzzle [30].

8.5.2 Results on Other Datasets

Since static pattern databases resulted in faster runtimes than dynamic pattern databases, we compared the algorithms with a static pattern database to the original heuristic functions on all the datasets. We used the grouping of $\lceil \frac{n}{2} \rceil - \lfloor \frac{n}{2} \rfloor$ for the static pattern databases on all the datasets, where n is the number of variables. The results are shown in Table 8.4.

For the BFBnB algorithm, the static pattern database reduced the number of nodes expanded by up to 5 times on some databases. The improvements were more modest on others, though. There are several explanations for the limited improvement on those datasets.

Table 8.3

A comparison of BFBnB and A* with various heuristics on $Auto$ and $Flag$.

Pattern Database			BFBnB		A*	
Dataset	Type	Size	Time (s)	Expanded	Time (s)	Expanded
Autos	Simple	26	461	62,721,601	674	35,329,016
Auto	Dynamic, k=2	41	449	52,719,793	148	6,286,142
Auto	Dynamic, k=3	116	468	49,271,809	76	2,829,877
Auto	Dynamic, k=4	582	699	48,057,205	67	2,160,515
Auto	Static, 9-9-8	1,280	495	57,002,715	228	9,763,518
Auto	Static, 13-13	16,384	211	48,814,334	125	4,762,276
Flag	Simple	29	OT	OT	OM	OM
Flag	Dynamic, k=2	45	1,222	132,431,610	824	19,359,296
Flag	Dynamic, k=3	149	788	79,332,390	207	5,355,085
Flag	Dynamic, k=4	858	1,624	84,054,443	350	7,377,817
Flag	Static, 10-10-9	2,560	2,600	249,638,318	OM	OM
Flag	Static, 15-14	49,152	720	88,305,173	136	4,412,232

Size means the number of patterns stored. *Time* means the running time (in seconds). *Nodes* means the number of nodes expanded by the algorithms. *OT* means the algorithm fails to f nish within a 1-hour time limit set for this experiment. *OM* means the algorithm used up all the RAM.

First, the amount of pruning for BFBnB hangs heavily on the quality of the given upper bound. As described in Section 6.1, we use a tabu hill climbing algorithm with random restarts to f nd the upper bound for pruning. While this algorithm has been shown to have good performance on many datasets, it offers no quality guarantees. Therefore, many extra nodes may be expanded because of the quality of the initial bound. Furthermore, as Vidal *et al.* [94] point out, some search problems are “easy”; others, because of characteristics of the particular dataset, are “hard”. (They mean “easy” or “hard” in the sense of relative diff culty, not in the sense that some are NP-hard and others are not.) In the case of “hard” datasets, even a good heuristic, such as our pattern databases, may not guide the search very well. Additionally, some of the datasets may be “easy” because the original heuristic is already tight. In these cases, the pattern databases do not improve the already tight bound.

The benef ts of the new techniques are more obvious when applied to the A* algorithm. For the datasets on which the original A* algorithm was able to f nish, the improved A* was up to one order of magnitude faster; the number of expanded nodes is also signif cantly reduced. In addition, the improved A* was able to solve three other datasets: Sensor Readings, Autos, and Flag. The running time on each of those datasets is pretty short, which indicates that once the memory consumption of the parent graphs was reduced, the A* algorithm was able to use more memory for the order graph and solved the search problems pretty easily.

Table 8.4

A comparison of BFBnB and A* on several datasets using static pattern databases.

Dataset			Results				
Name	n	N		BFBnB	BFBnB (SP)	A*	A* (SP)
Hepatitis	20	126	Time (s)	9	1	6	0
			Nodes	610,974	129,889	411,150	8,565
Parkinsons	23	195	Time (s)	100	19	100	15
			Nodes	8,388,607	4,646,877	8,388,607	1,152,576
Robot	25	5,456	Time (s)	632	3,121	OM	731
			Nodes	33,554,431	33,554,430	OM	3,286,650
Auto	26	159	Time (s)	1,170	211	OM	111
			Nodes	53,236,395	48,814,295	OM	4,762,276
Horse	28	300	Time (s)	4,221	678	OM	OM
			Nodes	268,435,455	74,204,000	OM	OM
Steel	28	1,941	Time (s)	7,913	4,544	OM	OM
			Nodes	268,435,455	264,887,347	OM	OM
Flag	29	194	Time (s)	12,902	421	OM	147
			Nodes	354,388,170	88,305,173	OM	4,412,232
WDBC	31	569	Time (s)	93,382	26,196	OM	OM
			Nodes	1,353,762,809	273,746,036	OM	OM

For the static pattern databases, groupings were $\lceil \frac{n}{2} \rceil - \lfloor \frac{n}{2} \rfloor$, where n is the number of variables, and sparse representation of parent scores (denoted by *SP*) against the original versions of these algorithms. n is the total number of variables. N is the number of data points.

CHAPTER 9

BOUNDED ERROR, ANYTIME, PARALLEL SEARCH

All of the algorithms presented so far execute serially. However, modern workstations often include 4, 8 and even up to 16 cores. Furthermore, manufacturers are rapidly approaching the physical barriers of how small they can produce microchips that behave reliably. Additionally, traditional shared memory architecture supercomputers are gradually being replaced by more cost-efficient, distributed memory clusters. According to the TOP500 list of fastest supercomputers in the world based on the HPL benchmark, the fastest three supercomputers (as well as many others) use a distributed memory model. For example, the K computer, currently ranked the fastest supercomputer, only allocates 16 GB of RAM for each core and offers no shared memory [97]. Message passing is necessary to exchange information between the cores.

Tamada *et al.* [88] have developed a parallel structure learning algorithm based on dynamic programming in the forward order graph using full parent graphs. In particular, in their distributed memory algorithm, they minimize the amount of communications required between processors by maximizing the overlap between subsets calculated at each processor. They begin with the observation that calculating $Score(\mathbf{U})$ requires $Score(\mathbf{U} \setminus \{X\})$ for all $X \in \mathbf{U}$. For example, suppose $\mathbf{A} = \{X_0, X_1, X_2\}$. Then calculating $Score(\mathbf{A})$ requires $Score(\{X_0, X_1\}), Score(\{X_0, X_2\})$ and $Score(\{X_1, X_2\})$. Further,

suppose that $\mathbf{B} = \{X_0, X_2, X_3\}$ and $\mathbf{C} = \{X_0, X_1, X_3\}$. So calculating $Score(\mathbf{B})$ requires $Score(\{X_0, X_2\})$, $Score(\{X_0, X_3\})$ and $Score(\{X_1, X_3\})$, and calculating $Score(\mathbf{C})$ requires $Score(\{X_0, X_1\})$, $Score(\{X_0, X_3\})$ and $Score(\{X_1, X_3\})$. Therefore, if a particular processor has the 5 necessary $Score(\cdot)$ values from layer 2, then it can reuse them to calculate all 3 new $Score(\cdot)$ values for layer 3. Sets that have many variables in common can reuse more scores than those that do not. On the other hand, consider $\mathbf{D} = \{X_4, X_5, X_6\}$. None of the earlier scores necessary for its calculation overlap those of $\mathbf{A}$. The intuition of their algorithm is to group sets with many overlapping variables on the same processor. They propose an indexing function which partitions variables in such a manner that provably maximizes the overlap among sets at the same processor. Consequently, it also provably minimizes the communication overhead and redundant communication. A key shortcoming of this parallel algorithm is its lack of anytime behavior. As with other dynamic programming algorithms [68, 84, 82, 59], this algorithm does not output any network until outputting the best network at the end of the search. The authors also note that, for large networks, despite minimizing communication, their MPI communication time still accounted for over 80% of the runtime.

9.1 Parallel Best-First Search

The heuristic search community has also developed a number of search algorithms that incorporate parallelism in a variety of ways. Many of those algorithms are based on best-f rst search.

9.1.1 Parallel Window Search

Parallel window search [76] is one of the oldest parallel search algorithms. It is an extension of iterative deepening A* (IDA*) [50]. IDA* is a limited-memory version of the A* algorithm in which the algorithm is given a threshold t. A normal depth-first search is then started from the start node; however, nodes whose f-cost exceeds t are pruned. If no goal node is found, the search begins again with a larger value of t. This process continues until a goal node is found. Asymptotically, IDA* expands the same number of nodes as A* for a tree search [50]. In practice, though, the search iterations which do not find a goal node can be time consuming [76] because each iteration is carried out serially.

The parallel window search algorithm distributes the execution of a number of IDA* processes, each with a different threshold, to different cores. In this manner, the running time of the parallel algorithm is only dependent on the time of the IDA* process with the smallest threshold that includes a goal node. If a process completes without finding a goal node, it restarts the search using a higher threshold than any of the other processes.

9.1.2 Parallel Retracting A*

Parallel retracting A* (PRA*) [28] and hash distributed A* (HDA*) [45] also extend best-first heuristic search to multiple cores. In these algorithms, a hash function is used to assign each node in a search space to a process. As a simple example, we could represent a node in the order graph using a bit vector in which the presence of a set bit indicates the respective variable is present in the subset and treat the resulting bit vector as a numeric data type (e.g., `long`). So we could represent the subset $\{X_0, X_3\}$ as the bit vector $\{1, 0, 0, 1\}$

which corresponds to the number 9. For our hash function, we could take the modulus of the number and the number of processes. If we had four processors, then $\{X_0, X_3\}$ would be mapped to processor $9\%4 = 1$. Each processor has its own open and closed lists which contain only nodes which the hash function maps to it. In parallel, each processor expands a node, uses the hash function to determine where to send all of the successors and uses a message passing scheme to send the successors to the appropriate processors. PRA* synchronously expanded nodes, so as soon as it would expand a goal state, the search ceases. The synchronicity introduces overhead, however. To address that overhead, HDA* used asynchronous message passing. However, with this strategy, expanding a node does not necessarily mean the best path to it has been found. For example, a better predecessor could be "in transit". Consequently, HDA* may need to re-expand nodes. Similarly, after expanding a goal state, it must ensure that no better paths were available but had not yet been expanded because of the non-determinism introduced by parallel execution.

9.1.3 Adaptive k-Parallel Best-First Search

Adaptive k-parallel best-first search [94] is another approach to parallelize A* search. It is a parallel adaption of the k best-first search (KBFS) algorithm [31] for multi-core, shared memory architectures. The sequential version of KBFS proceeds much like a typical best-first search algorithm; however, at each step, rather than expanding the single best node, the k best nodes are expanded. Their successors are added to the open list, and the next iteration of the algorithm begins. The parallel version of KBFS observes that this process is easily parallelized by expanding the best k nodes in separate processes. In their

implementation of parallel KBFS, Vidal *et al.* [94] assume only one open and closed list (each) exist in shared memory. Therefore, access to these data structures is regarded as a *critical section* of the code; that is, only a single process can modify the data structures at once. Consequently, access to the open and closed lists is a bottleneck for their algorithm. Based on that, the authors suggest that, like KBFS, their algorithm is more useful when node expansions are expensive. They also focus on sub-optimal planning using a non-admissible heuristic.

Initial experiments revealed that many problems did not benef t from the parallelism. The authors observe that many of these problems are "easy", while others that do benef t from the parallelism are "hard." (This is only in the sense of relative diff culty, not that some of the problems are NP-hard and others are not.) The overhead associated with parallelism often trumps any benef ts for the "easy" problems. Based on these observations, the authors devised a scheme in which the number of threads is increased as the algorithm determines that a problem is "hard." They assume that node expansions determine the diff culty of the problem. For up to 50 node expansions, only a single thread is used. Four threads are used for up to 400 nodes, 8 for 3,000 nodes, 16 threads for up to 20,000 nodes, 32 threads up to 100,000 nodes, and 64 threads are used for the remainder of the search. The authors also observed that restarting the search after increasing the number of threads improved the diversity of nodes expanded, which is important for sub-optimal search.

9.1.4 Parallel Structured Duplicate Detection and Parallel Best-NBlock First Search

Parallel structured duplicate detection (PSDD) [107] and Parallel Best-NBlock-First (PBNF) [7] parallelize the structured duplicate detection (SDD) algorithm [105]. In SDD, a projection function, p is used to map a concrete state of a state space into an abstract state in an abstract state space. An $n$block is the set of all nodes that map to the same abstract state. A node x' is a predecessor of y' in the abstract state space iff there exist nodes x and y in the original state space such that x is a predecessor of y and $p(x) = x'$ and $p(y) = y'$. The *duplicate detection scope* of a node x in the original state space is all y' in the abstract state space such that y' is a successor of $p(x)$. When expanding x, only nodes in its duplicate detection scope need to be checked for duplicates. This generalizes to all nodes in the $n$block given by $p(x)$. Originally, SDD used this strategy to reduce the RAM requirements for breadth-first search. In particular, when expanding nodes in a particular $n$block, only its successors in the abstract state space need to be in RAM at once.

PSDD adds parallelization to SDD. In particular, if the duplicate detection scope of two $n$blocks does not overlap, they can be expanded at the same time without risk of generating successors in the same $n$block. The algorithm uses a single lock on the abstract state graph to indicate which $n$blocks are being expanded or in the duplicate detection scope of another abstract state being expanded. An $n$block can be expanded when neither it nor anything in its duplicate detection scope is used by another process. PSDD expands nodes in a breadth-first order. PBNF also adopts SDD, but expands nodes in a best-first order. Similar locking mechanisms are used as in PSDD; however, a data structure is also used to indicate the lowest f value of a node in each free $n$block (i.e., one that is not being expanded or in the

duplicate detection scope of an $n$block that is being expanded). A processor expands nodes in its current $n$block until it encounters a node with a higher f value than the lowest f value of a free n block. The process will then expand nodes from the new $n$block. Speculative expansion is used to ensure processors are not idle and do not incur too much overhead swapping between $n$blocks.

9.1.5 Parallel Frontier A* with Delayed Duplicate Detection

Frontier A* (FA*) search [49] is an approach to best-first search in which a closed list is not used. Rather, only the open list is kept in RAM for duplicate detection. Each node in the open list is annotated with a set of *used operator bits* indicating which of its neighbors have already been expanded. Nodes are expanded in a best-first order, so if a neighbor of a node has already been expanded, there is no need for it to be re-generated. As described in Section 6.4, delayed duplicate detection [51] is a strategy in which external memory is used to store nodes generated at a particular layer of a search. External-memory sorting, such as merge sort [37], is used to sort nodes and remove duplicates. Niewiadomski *et al.* [66] present an algorithm the incorporates both FA* and DDD. They address the "leak back problem" [51] in which nodes may be re-expanded by using two types of closed lists. The ClosedIn list maintains edges from non-closed to closed nodes, while the ClosedOut list maintains edges from closed to non-closed nodes. Generated nodes are added to the Open list if they are not in ClosedIn.

To parallelize FA*-DDD, they also assign an integer to each node. Nodes are distributed to processors according to their integer values. The parallel algorithm consists

of f ve phases. In the f rst, processors communicate the number of nodes which have the global minimum f value and transfer nodes based on their integer values. Next, the processors expand their nodes which have the global minimum f value. Third, the algorithm determines how to distribute nodes based on the range of integer values. Then, the processors determine the next lowest global minimum f value. Finally, information from the previous iteration is deleted and the algorithm begins with the f rst step again.

9.1.6 Parallel Dovetailing

Parallel dovetailing [92] is another parallel search technique in the same vein as parallel window search. Valenzano *et al.* [92] observe that many sub-optimal search algorithms require some sort of parameter conf guration. For example, IDA* requires the threshold, weighted A* requires the weight to use, beam search requires the size of beam to use and KBFS requires k. Adaptive parallel KBFS showed a method by which k is updated throughout the search. In contrast, parallel dovetailing begins by selecting a variety of parameter conf gurations and running each conf guration at the same time in parallel. Thus, parallel window search is a special case of parallel dovetailing specif c for IDA* and considering only the threshold as a parameter. The authors point out, though, that other, more subtle design decisions can affect the algorithm performance. For example, the order in which successor generation operators are applied can greatly impact the runtime behavior of an algorithm. As presented, each algorithm conf guration runs in parallel until any of them reach a solution, regardless of its optimality. At that point, a message is broadcast to

all running tasks that a solution has been found and the search can stop. Because of this behavior, the version of parallel dovetailing presented is applicable to sub-optimal search.

9.2 BEAP Search Algorithm

Based on the limitations of the parallel dynamic programming algorithm of Tamada *et al.* and the results of parallel best-first heuristic search, we developed a bounded error, anytime, parallel (BEAP) search algorithm. This algorithm is an example of parallel dovetailing [92] using WA* (see Section 7.1.1). In this algorithm we select a range of ϵ values and run one WA* process for each value in parallel. We adapt the A* algorithm in Table 5.1 into WA* by passing ϵ as an additional input to the algorithm. The only change required to the algorithm is that, when calculating h in Line 20, we multiply the value by ϵ. This works with both the simple heuristic given in Equation 5.1 as well as with pattern databases. There is no communication between the processes, so they do expand some of the same nodes.

The anytime behavior of the parallel algorithm results because, as the WA* instances complete, their solutions give an upper bound on the optimal score of the Bayesian network. Typically, instances with large ϵ values finish very quickly, but the scores of the learned network are high (always bounded by ϵ, though). Instances with lower ϵ values finish more slowly, but have better scores. Therefore, as the search progresses and WA* instances complete, the upper bound improves. Finally, the completion of an instance in which $\epsilon = 1$, which we denote as ϵ_1, gives the provably optimal network.

As each instance of WA* completes, the quality of the solution is bounded by ϵ. Consequently, as more instances complete, the provable bound between the optimal network and the best learned network decreases. Running ϵ_1 offers another way to calculate a bound on the error, though. Because ϵ_1 does not weight the heuristic, it is guaranteed to be admissible since we use admissible heuristics in our search. Coupled with the best-first expansion policy, no optimal network could possibly have a score better than the f cost of the most recently expanded node of ϵ_1, so that serves as a lower bound on the optimal network score. Also, that lower bound is guaranteed to increase (or stay the same) with each node expanded in ϵ_1 because of the best-first expansion. Therefore, the ratio between the score of the best learned network and the f cost of the most recently expanded node of ϵ_1 gives another bound on the solution quality. As shown in Section 9.4, the ratio bound is often tighter than the bound guaranteed by ϵ of the other instances of WA*.

9.3 Advantages of BEAP

The BEAP algorithm has several advantages compared to other parallel Bayesian network structure learning algorithms. First, it has very little communication overhead because each WA* process uses a different ϵ; the processes do not communicate. The limited communication ensures that runtime is not wasted passing messages or waiting for synchronization, which plagued the parallel DP algorithm [88]. Second, a proper range of ϵ_is gives the parallel algorithm very good anytime behavior. The parallel DP algorithm [88] does not have anytime behavior at all.

BEAP also has some similarites to, and advantages over, several serial anytime search algorithms, including AWA* [41] and Anytime Repairing A*(ARA*) [58]. All three algorithms use a weighted heuristic to provably bound the error of solutions. BEAP offers advantages over these serial anytime search algorithms, though. First, BEAP re-expands nodes in parallel rather than serially. Second, in order to calculate a tighter bound than that given by ϵ, AWA* and ARA* must search through the open list and calculate the true f value of each node. In constrast, BEAP simply uses the f value of the most recently expanded node of ϵ_1. Third, unlike ARA*, BEAP does not require any data structures other than those normally required by A*. Like AWA* and ARA*, though, BEAP is a general purpose search algorithm that could be applied to any heuristic search problem, not just structure learning.

9.4 Experimental Results

We evaluated BEAP on a set of benchmark datasets from the UCI repository [33]. For all datasets, we removed records with missing values and discretized all variables into two states. The experiments were performed on a PC with 3.07 GHz Intel i7 processor and 16 GB of RAM. We compared BEAP to BB and a custom implementation of AWA*. The AWA* implementation is a straight-forward adaptation of the existing A* algorithm [102]. Even though they are anytime algorithms, we did not compare to any local search algorithms because they do not give an error bound. For BEAP, we used four different values of ϵ: 1.2, 1.08, 1.04 and 1. We empirically determined that $\epsilon > 1.2$ did not improve learning. We allowed all algorithms a total execution time of 30 minutes, not including

local score calculations. BB and AWA* are sequential, so we gave them 30 minutes of wall clock time. Since BEAP used four processes (one for each value of ϵ), we gave it 7.5 minutes of wall clock time, so its total time was also 30 minutes. Each BEAP process had 4 GB of RAM.

9.4.1 Node Expansions

We first evaluated the number of nodes expanded by BEAP for each value of ϵ. The results in Figure 9.1 show that the algorithm typically found high quality solutions quickly. The figure also sheds insight into several characteristics of the search algorithm.

First, the searches with high ϵ, usually expand a very small number of nodes. For example, on five of the datasets, the process with $\epsilon = 1.2$ expands the minimum number of nodes possible to find a solution ($n + 1$). This takes only a fraction of a second; that processor is idle for the rest of the search. This behavior suggests that a scheme similar to that in parallel window search [76] could be used to more fully utilize the available resources. In particular, that processor could then begin a search with a weight of, for example, 1.06. If another processor finished, it could search with a weight of 1.03. Completion of these searches would give tighter bounds.

Second, the figure suggests that, like other combinatorial optimization problems, structure learning has a *critical point* [103]. A critical point for a problem is a point at which the problem difficulty undergoes a major change. For example, the problem of finding an optimal path to a goal node in a random tree is polynomial if the probability that any node has a zero-cost edge to a successor is greater than 1, but exponential otherwise [61]. Mov-

ing across this critical point is called a *phase transition*. Based on Figure 9.1, the critical point for structure learning appears to be between 8% and 4% of optimal. Nearly all of the instances for $\epsilon = 1.08$ complete quickly; however, over half fail for $\epsilon = 1.04$. These results indicate that f nding a network that is 8% of optimal is much easier than f nding one that is 4% of optimal.

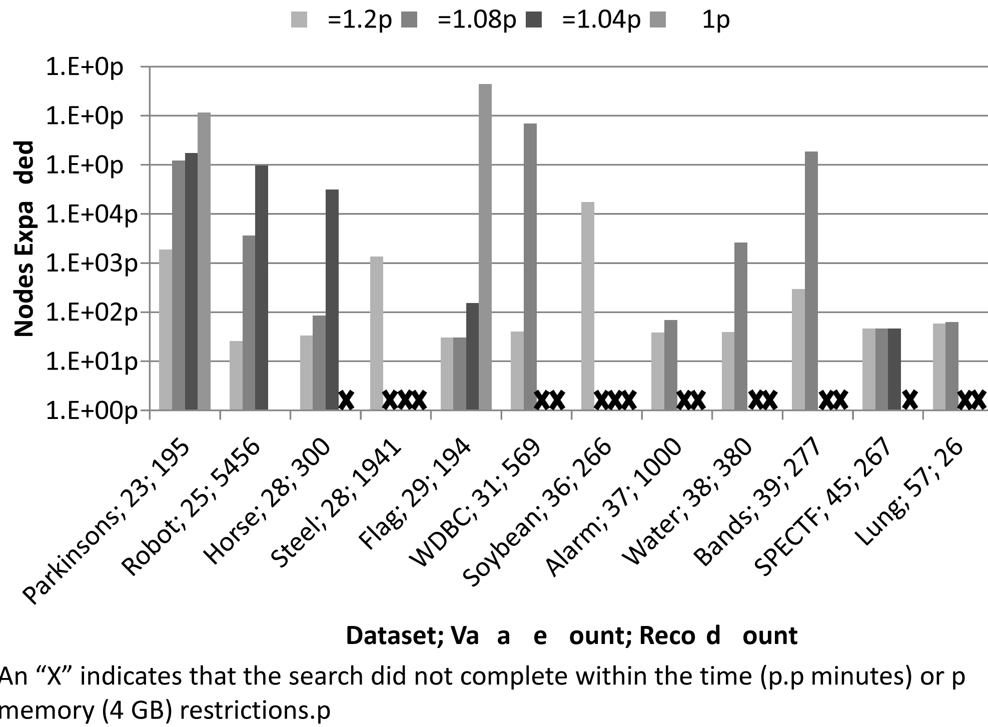


Figure 9.1

Order graph nodes expanded for each dataset and value of ϵ by BEAP.

9.4.2 Comparison of Anytime Behavior

We next compared the convergence and anytime behavior of BEAP compared to BB. As the convergence curves in Figure 9.2 show, BEAP f nds provably high quality solutions very quickly on all of the datasets. For both $Flag$ and $SPECTF$, within 2 seconds of wall clock time (8 seconds of total computing time), BEAP found networks with scores provably within 2.5% of optimal. The curves demonstrate that BEAP and BB improve error bounds differently. BB never improves its initial solution, but spends the entire 30 minutes improving its lower bound. As BEAP processes complete and ϵ_1 expands more nodes, both upper and lower bounds improve.

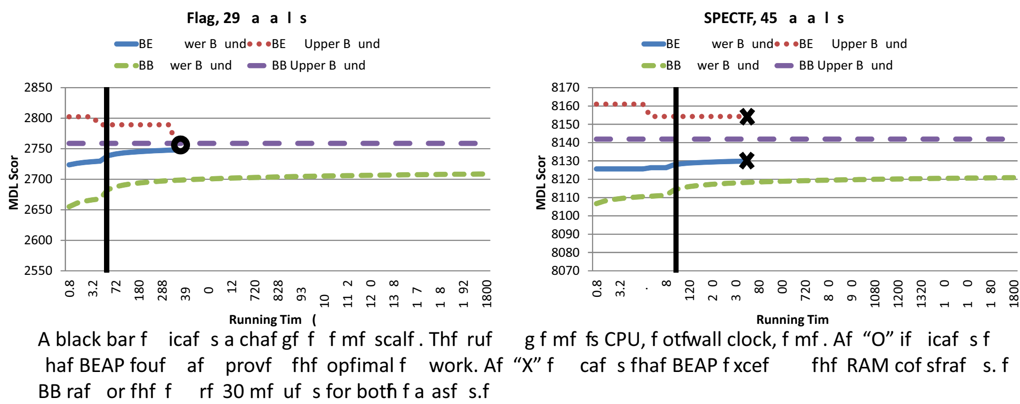


Figure 9.2

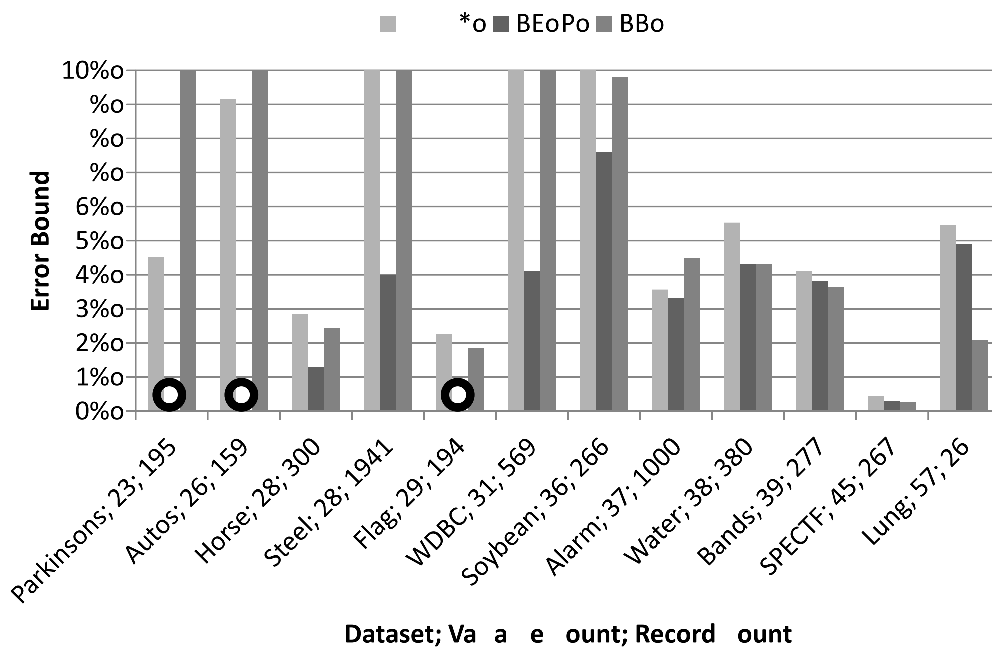
Convergence behavior or BEAP and BB.

9.4.3 Comparison of Solution Quality

Finally, we compared the solution quality of BEAP to AWA* and BB by comparing their upper and lower bounds. As Figure 9.2 shows, BEAP almost always finds a solution with a tighter error bound than the other algorithms. BEAP is the only algorithm which finds and proves the optimal structure on any of the datasets. It found tighter solutions than AWA* because BEAP never re-expands nodes within the same process; AWA* must re-expand a node each time it finds a better path to it. BB searches in the space of cyclic graphs, so these results suggest that the heuristic search formulation more effectively guides the algorithm to higher quality solutions than breaking cycles.

The bounds for BEAP are always better than the best ϵ_i that was solved (shown in Figure 9.1). This shows that the bound given by the ratio between ϵ_1 and the best solution is always tighter.

For all algorithms, these results compare very favorably to those for parallel DP [88]. That algorithm took 483,874 seconds to find the optimal network for a 32 variable dataset. Of that time, 392,186 seconds were spent in MPI communication. Their algorithm also required 836.1 GB of RAM. In contrast, our algorithm used at most 16 GB, and typically less than 8 GB, which is an improvement of nearly two orders of magnitude.



n “O” indicates that BEoP found and proved the optimal network within the resource o constraints.o

Figure 9.3

The solution quality of networks learned by AWA*, BEAP and BB.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have presented a novel heuristic graph search perspective for learning optimal Bayesian network structures. In the section, we review the contributions of this dissertation. We then describe some avenues for future work.

10.1 Contributions

We have made the following contributions:

- cast an existing dynamic programming formulation of structure learning into the context of heuristic graph search;
- formulated eff cient data structures and representations to calculate and store information necessary in the search;
- given a lower bound function that can be used to guide the heuristic search, thereby ignoring nodes the existing dynamic programming algorithms waste time and memory expanding and storing;
- shown how to effectively leverage the regular structure of the search graph to discard information once it is no longer necessary and use external memory when the problem size grows too large to f t into RAM;
- developed anytime algorithms that can both f nd good networks quickly and, given enough time, f nd provably optimal networks;
- improved upon the lower bound function using pattern databases to calculate much tighter bounds, which allow us to solve larger problems more quickly;
- demonstrated how simple parallel algorithms can quickly f nd provably high quality algorithms using orders of magnitude less resources than existing parallel optimal learning algorithms.

We tested all of our algorithms on a variety of commonly used machine learning benchmark datasets against current state of the art algorithms. In most cases, we showed that our algorithms outperformed existing methods by running faster, using less memory and f nding better solutions more quickly.

Improving the scalability of optimal structure learning algorithms has many practical applications. Learning regulatory networks is a very active area of research in computational biology, and our rigorously grounded learning methods can replace many of the ad-hoc programs currently in use [96, 9, 98, 77]. Optimal algorithms remove the uncertainty associated with structure learning and allow the biologists to focus on interpretting the results. Similarly, as discussed in Section 2.3.2, there are many choices for scoring functions when learning Bayesian networks. Optimal structure learning algorithms allow researchers to directly evaluate the merits of each scoring function by, for example, comparing a learned network to a gold standard network using structural hamming distance [91] or KL divergence [52].

10.2 Future Work

This work can be extended in several different ways.

Hybrid Search Techniques Throughout this dissertation, we have focused only on unconstrained score-based learning methods. That is, we always search for a network that optimizes the given scoring function; however, another class of algorithms known as constraint-based algorithms [85] are also used to learn Bayesian network structures. Constraint-based

algorithms begin with a set of conditional indepence tests to establish the relationships among the variables. Then, based on the results of the tests, edges are added to the network in a manner to satisfy as many of the tests as possible. Typically, constraint-based algorithms only require a polynomial number of tests. Compared to the exponential search space for score-based algorithms, this seems like an improvement. Unfortunately, the constraint-based searches are very susceptible to noisy and small datasets because they reduce the reliability of the independence tests.

Recently, several hybrid algorithms [91, 73, 47] have been proposed which incorporate elements of both constraint- and score-based methods. They begin with a set of conditional independence tests to establish a super-structure skeleton for the network. That is, edges in the super-structure are not directed, but only edges present in the super-structure may appear in the final network. The Max-Min Hill Climbing algorithm (MMHC) [91] then performs a greedy hill climbing search in the space restricted by the super-structure. As with any greedy hill climbing search, there are no quality guarantees for the learned network. The constrained optimal search (COS) [73] and ancestral constrained optimal search (ACOS) [47] also begin with a set of conditional independence tests to identify a super-structure. However, they then use dynamic programming to guarantee to find the optimal network that adheres to the super-structure.

The conditional independence tests and resulting super-structure can greatly reduce the size of the search space of possible network structures. As we have shown, though, our heuristic search algorithms outperform dynamic programming in a number of measures, including running time, memory usage and anytime behavior. We could easily apply our

algorithms to the space restricted by the super-structure. A more interesting extension could relate to the phase transitions discussed in Section 9.4.1. It is possible that a phase transition exists based on properties of the super-structure. Because the signif cance cutoffs of independence tests is always a user-supplied value, they could always be specif ed in a manner to keep the problem on the "easy" side of the critical points. Some early results [67] suggest that state spaces induced by super-structures have phases in which structure learning is linear if both treewidth and the maximum degree of the super-structure are bounded by arbitrary constants.

Expert Knowledge Super-structures are one way to introduce constraints into structure learning. For many f elds, such as computational biology, a massive amount of data is available which could potentially help in structure learning. For example, due to wet lab experiments, we may know that X should be a parent of Y. de Campos and Ji's branch and bound algorithm [19] can use simple constraints; however, no dynamic programming-based algorithms can currently take advantage of structure constraints. Incorporating these into structure learning should reduce both the time and memory requirements by pruning parts of the search inconsistent with the constraints.

Score Calculations Currently, all optimal structure learning algorithms assume all necessary local scores are pre-computed and easily accessible. We showed in Section 6.3 how to store the scores on disk if necessary and nodes are expanded in lexicographic order; however, because of the eff cient AD-tree-like search, we must store all scores in RAM at least during the score calculation phase. It may be possible to use a form of delayed

duplicate detection in which scores are periodically written to disk and summed together at the end of the search if they cannot all fit in RAM.

Tian [90] and de Campos and Ji [19] give results for pruning scores without needing to actually calculate them. In that sense, then, those pruning results are more helpful than Theorem 1 because it still requires the score be calculated before it can be pruned. Because we incrementally calculate scores using the AD-tree-like search, we cannot effectively take advantage of these results. An alternative to our incremental calculation strategy would be to store an actual AD-tree in memory and calculate the scores one at a time. de Campos and Ji's implementation adopts this approach. In practice, even though their implementation is in C++ and ours is in Java, the incremental calculation strategy significantly outperforms the one at a time strategy, even though it allows more pruning (see, for example, Section 5.5). The dynamic programming algorithm of Silander and Myllymaki [82] also uses an incremental calculation strategy.

However, a super-structure induced reduced space of networks would allow even more pruning, as would expert knowledge. Moore and Wong [64] use RADSEARCH [63] to make the score calculations for optimal reinsertion more efficient. A similar approach could allow more pruning during score caclulation rather than having to wait until after the scores are calculated to prune. This improvement could benefit all optimal structure learning algorithms since they all require local scores.

Publication Parts of this dissertation have been published in the the following conference papers: [102, 59, 60]. The rest of it has been submitted to either journals or conferences and is under review (as of June 7, 2012).

REFERENCES

[1] S. Aine, P. P. Chakrabarti, and R. Kumar, "AWA*-a window constrained anytime heuristic search algorithm," *Proceedings of the 20th international joint conference on Artif cal intelligence*, San Francisco, CA, USA, 2007, IJCAI'07, pp. 2250–2255, Morgan Kaufmann Publishers Inc.

[2] H. Akaike, "Information Theory and an Extension of the Maximum Likelihood Principle," *Proceedings of the Second International Symposium on Information Theory*, 1973, pp. 267–281.

[3] O. Barriàre, E. Lutton, and P.-H. Wuillemin, "Bayesian network structure learning using cooperative coevolution," *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, New York, NY, USA, 2009, GECCO '09, pp. 755–762, ACM.

[4] M. Boddy and T. Dean, "Solving time-dependent planning problems," *Proceedings of the 11th international joint conference on Artif cial intelligence - Volume 2*, San Francisco, CA, USA, 1989, IJCAI'89, pp. 979–984, Morgan Kaufmann Publishers Inc.

[5] H. Bozdogan, "Model selection and Akaike's Information Criterion (AIC): The general theory and its analytical extensions," *Psychometrika*, vol. 52, 1987, pp. 345–370, 10.1007/BF02294361.

[6] W. Buntine, "Theory ref nement on Bayesian networks," *Proceedings of the seventh conference (1991) on Uncertainty in artif cial intelligence*, San Francisco, CA, USA, 1991, pp. 52–60, Morgan Kaufmann Publishers Inc.

[7] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-First Heuristic Search for Multicore Machines," *Journal of Artif cial Intelligence*, vol. 39, 2010, pp. 689–743.

[8] R. Castelo and A. Siebes, "Priors on network structures. Biasing the search for Bayesian networks," vol. 24, 2000, pp. 39–57–.

[9] X.-w. Chen, G. Anantha, and X. Wang, "An effective structure learning method for constructing gene networks," *Bioinformatics*, vol. 22, no. 11, 2006, pp. 1367–1374.

[10] D. M. Chickering, “A Transformational Characterization of Equivalent Bayesian Network Structures,” *Proceedings of the Eleventh Conference on Uncertainty in Artif cial Intelligence*, 1995, pp. 87–98.

[11] D. M. Chickering, “Learning Bayesian Networks is NP-Complete,” *Learning from Data: Artif cial Intelligence and Statistics V*. 1996, pp. 121–130, Springer-Verlag.

[12] C. Chow and C. Liu, “Approximating discrete probability distributions with dependence treesID - 196,” 1968.

[13] G. F. Cooper and E. Herskovits, “A Bayesian Method for the Induction of Probabilistic Networks from Data,” *Mach. Learn.*, vol. 9, October 1992, pp. 309–347.

[14] J. C. Culberson and J. Schaeffer, “Pattern Databases,” vol. 14, 1998, pp. 318–334.

[15] J. Cussens, “Bayesian network learning with cutting planes,” *Proceedings of the Proceedings of the Twenty-Seventh Conference Annual Conference on Uncertainty in Artif cial Intelligence (UAI-11)*, Corvallis, Oregon, 2011, pp. 153–160, AUAI Press.

[16] R. Daly and Q. Shen, “Learning Bayesian network equivalence classes with Ant Colony optimization,” *J. Artif. Int. Res.*, vol. 35, June 2009, pp. 391–447.

[17] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*, Cambridge University Press, 2009.

[18] C. P. De Campos and Q. Ji, “Properties of Bayesian Dirichlet Scores to Learn Bayesian Network Structures,” *Twenty-Fourth AAAI Conference on Aritif cial Intelligence (AAAI-10)*, 2010.

[19] C. P. de Campos and Q. Ji, “Eff cient Learning of Bayesian Networks using Constraints,” *Journal of Machine Learning Research*, vol. 12, 2011, pp. 663–689.

[20] C. P. de Campos and J. Qiang, “Improving Bayesian Network parameter learning using constraints,” *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, 2008, pp. 1–4–.

[21] C. P. de Campos, Z. Zeng, and Q. Ji, “Structure learning of Bayesian networks using constraints,” *Proceedings of the 26th Annual International Conference on Machine Learning*, New York, NY, USA, 2009, ICML ’09, pp. 113–120, ACM.

[22] L. M. de Campos and J. G. Castellano, “Bayesian network learning algorithms using structural restrictions,” vol. 45, 2007, pp. 233–254–.

[23] L. M. de Campos and J. F. Huete, “A new approach for learning belief networks using independence criteria,” *International Journal of Approximate Reasoning*, vol. 24, no. 1, 2000, pp. 11 – 37.

[24] T. Dean and M. Boddy, "An Analysis of Time-Dependent Planning," *Proceedings*, 1988.

[25] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data Via the EM Algorithm," 1977, pp. 1–22–.

[26] G. Elidan and N. Friedman, "Learning Hidden Variable Networks: The Information Bottleneck Approach," vol. 6, 2005, pp. 81–127–.

[27] G. Elidan, N. Lotner, N. Friedman, and D. Koller, "Discovering Hidden Variables - A Structure-Based Approach," *Neural Informatin Processing Systems 13*. 2000, pp. –, MIT Press.

[28] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "PRA*: a memory-limited heuristic search procedure for the Connection Machine," *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, oct 1990, pp. 145 –149.

[29] A. Feelders and L. C. van der Gaag, "Learning Bayesian network parameters under order constraints," *Int. J. Approx. Reasoning*, vol. 42, May 2006, pp. 37–53.

[30] A. Felner, R. Korf, and S. Hanan, "Additive Pattern Database Heuristics," *Journal of Artif cial Intelligence Research*, vol. 22, 2004, pp. 279–318.

[31] A. Felner, S. Kraus, and R. E. Korf, "KBFS: K-Best-First Search," *Annals of Mathematics and Artif cial Intelligence*, vol. 39, no. 1, 2003, pp. 19–39.

[32] B. Fitelson, "Likelihoodism, Bayesianism, and relational conf rmation," *Synthese*, vol. 156, 2007, pp. 473–489, 10.1007/s11229-006-9134-9.

[33] A. Frank and A. Asuncion, "UCI Machine Learning Repository," 2010.

[34] N. Friedman, "Learning Belief Networks in the Presence of Missing Values and Hidden Variables," 1997.

[35] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classif ers," *Mach. Learn.*, vol. 29, November 1997, pp. 131–163.

[36] N. Friedman, I. Nachman, and D. Peer, "Learning Bayesian network structure from massive datasets: The "sparse candidate" algorithm," *Proceedings of UAI-13*, 1999, pp. 206–215.

[37] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2 edition, Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[38] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[39] Z. Ghahramani, "Learning Dynamic Bayesian Networks," *Adaptive Processing of Sequences and Data Structures, International Summer School on Neural Networks, "E.R. Caianiello"-Tutorial Lectures*, London, UK, 1998, pp. 168–197, Springer-Verlag.

[40] F. Glover, "Tabu Search: A Tutorial," *Interfaces*, vol. 20, no. 4, July/August 1990, pp. 74–94.

[41] E. A. Hansen and R. Zhou, "Anytime Heuristic Search," *Journal of Artif cial Intelligence Research*, vol. 28, 2007, pp. 267–297.

[42] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Ieee Transactions On Systems Science And Cybernetics*, vol. 4, no. 2, 1968, pp. 100–107.

[43] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian networks: The combination of knowledge and statistical data," vol. 20, 1995, pp. 197–243.

[44] T. Jaakkola, D. Sontag, A. Globerson, and M. Meila, "Learning Bayesian Network Structure using LP Relaxations," *Proceedings of the 13th International Conference on Artif cial Intelligence and Statistics (AISTATS)*, 2010.

[45] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning," *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009*, Thessaloniki, Greece, 2009.

[46] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicles 0-4*, 1st edition, Addison-Wesley Professional, 2009.

[47] K. Kojima, E. Perrier, S. Imoto, and S. Miyano, "Optimal Search on Clustered Structural Constraint for Learning Bayesian Network Structure," vol. 11, pp. 285–310.

[48] P. Kontkanen and P. Myllymäki, "A linear-time algorithm for computing the multinomial stochastic complexity," *Inf. Process. Lett.*, vol. 103, September 2007, pp. 227–233.

[49] R. Korf, W. Zhang, I. Thayer, and H. Hohwald, "Frontier search," *Journal of the ACM*, vol. 52, no. 5, 2005, pp. 715–748.

[50] R. E. Korf, "Depth-f rst iterative-deepening: an optimal admissible tree search," *Artif cial Intelligence*, vol. 27, 1985, pp. 97–109.

[51] R. E. Korf, "Best-f rst frontier search with delayed duplicate detection," *Proceedings of the 19th national conference on Artif cal intelligence*, San Jose, California, 2004, pp. 650–657, AAAI Press.

[52] S. Kullback and R. Leibler, "On Information and Suff ciency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, March 1951, pp. 79 – 86.

[53] W. Lam and F. Bacchus, "Using Causal Information and Local Measures to Learn Bayesian Networks," *Proceedings of Uncertainty in Artif cial Intelligence*, 1993, pp. 243–250.

[54] W. Lam and F. Bacchus, "Learning Bayesian belief networks: An approach based on the MDL principle," *Computational Intelligence*, vol. 10, 1994, pp. 269–293.

[55] P. Langley, W. Iba, and, and K. Thompson, "An analysis of Bayesian classif ers," *Proceedings of the tenth national conference on Artif cial intelligence*. 1992, AAAI'92, pp. 223–228, AAAI Press.

[56] P. Larranaga, M. Poza, Y. Yurramendi, R. H. Murga, and C. M. H. Kuijpers, "Structure learning of Bayesian Networks by GenetioAlgorithms: A performance analysis of control parameters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1996, pp. 912–926.

[57] S. L. Lauritzen and D. J. Spiegelhalter, *Local computations with probabilities on graphical structures and their application to expert systems*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990, pp. 415–448.

[58] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* Search with Provable Bounds on Sub-Optimality," *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, S. Thrun, L. Saul, and B. Schölkopf, eds. 2003, MIT Press.

[59] B. Malone, C. Yuan, and E. Hansen, "Memory-Eff cient Dynamic Programming for Learning Optimal Bayesian Networks," *Proceedings of the 25th national conference on Artif cal intelligence*, 2011.

[60] B. Malone, C. Yuan, E. Hansen, and S. Bridges, "Improving the Scalability of Optimal Bayesian Network Learning with External-Memory Frontier Breadth-First Branch and Bound Search," *Proceedings of the Proceedings of the Twenty-Seventh Conference Annual Conference on Uncertainty in Artif cial Intelligence (UAI-11)*, Corvallis, Oregon, 2011, pp. 479–488, AUAI Press.

[61] C. McDiarmid, "Probabilistic Analysis of Tree Search," *Disorder in Physical Systems*. 1990, pp. 249–260, Oxford Science.

[62] A. Moore and M. S. Lee, "Cached suff cient statistics for eff cient machine learning with large datasets," *J. Artif. Int. Res.*, vol. 8, March 1998, pp. 67–91.

[63] A. Moore and J. Schneider, "Real-valued All-Dimensions search: Low-overhead rapid searching over subsets of attributes," *Proceedings of the Eighteenth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-02)*, San Francisco, CA, 2002, pp. 360–369, Morgan Kaufmann.

[64] A. Moore and W.-K. Wong, "Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning," *Intl. Conf. on Machine Learning*, 2003, pp. 552–559.

[65] K. P. Murphy, *Dynamic bayesian networks: representation, inference and learning*, doctoral dissertation, 2002, AAI3082340.

[66] R. Niewiadomski, J. N. Amaral, and R. C. Holte, "Sequential and parallel algorithms for frontier A* with delayed duplicate detection," *proceedings of the 21st national conference on Artificial intelligence - Volume 2*. 2006, AAAI'06, pp. 1039–1044, AAAI Press.

[67] S. Ordyniak and S. Szeider, "Algorithms and Complexity Results for Exact Bayesian Structure Learning," *Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, Corvallis, Oregon, 2010, pp. 401–408, AUAI Press.

[68] S. Ott, S. Imoto, and S. Miyano, "Finding Optimal Models for Small Gene Networks," *Pac. Symp. Biocomput*, 2004, pp. 557–567.

[69] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[70] P. Parviainen and M. Koivisto, "Exact structure discovery in Bayesian networks with less space," *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, Montreal, Quebec, Canada, 2009, AUAI Press.

[71] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[72] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*, Morgan Kaufmann Publishers Inc., 1988.

[73] E. Perrier, S. Imoto, and S. Miyano, "Finding Optimal Bayesian Network Given a Super-Structure," *Journal of Machine Learning Research*, vol. 9, October 2008, pp. 2251–2286.

[74] I. Pohl, "First Results on the Effect of Error in Heuristic Search," 1969, pp. 219–236, cited By (since 1996) 6; Conference of Machine Intelligence 5, Annu Mach Intel Workshop.

[75] I. Pohl, "Heuristic search viewed as path f nding in a graph," *Artif cial Intelligence*, vol. 1, no. 3-4, 1970, pp. 193 – 204.

[76] C. Powley and R. E. Korf, "Single-agent parallel window search: a summary of results," *Proceedings of the 11th international joint conference on Artif cial intelligence - Volume 1*, San Francisco, CA, USA, 1989, IJCAI'89, pp. 36–41, Morgan Kaufmann Publishers Inc.

[77] Y. Qi, Y. Zhang, J. Lv, H. Liu, J. Zhu, and J. Su, "Deducing Causal Relationships among Different Histone Modif cations, DNA Methylation and Gene Expression," *Proceedings of the 2009 Fifth International Conference on Natural Computation - Volume 06*. 2009, pp. 139–143, IEEE Computer Society.

[78] J. Rissanen, "Fisher information and stochastic complexity," *Information Theory, IEEE Transactions on*, vol. 42, no. 1, jan 1996, pp. 40 –47.

[79] T. Schlitt and A. Brazma, "Current approaches to gene regulatory network modelling," *BMC Bioinformatics*, vol. 8, no. Suppl 6, 2007, p. S9.

[80] G. Schwarz, "Estimating the Dimension of a Model," vol. 6, 1978, pp. 461–464.

[81] T. Silander, P. Kontkanen, and P. Myllymaki, "On Sensitivity of the MAP Bayesian Network Structure to the Equivalent Sample Size Parameter," *Proceedings of the Twenty-Third Conference Annual Conference on Uncertainty in Artif cial Intelligence (UAI-07)*, Corvallis, Oregon, 2007, pp. 360–367, AUAI Press.

[82] T. Silander and P. Myllymaki, "A simple approach for f nding the globally optimal Bayesian network structure," *Proceedings of the 22nd Annual Conference on Uncertainty in Artif cial Intelligence (UAI-06)*, Arlington, Virginia, 2006, AUAI Press.

[83] T. Silander, T. Roos, P. Kontkanen, and P. Myllymaki, "Factorized normalized maximum likelihood criterion for learning Bayesian network structures," *Proceedings of the 4th European Workshop on Probabilistic Graphical Models (PGM-08)*, 2008, pp. 257–272.

[84] A. Singh and A. Moore, *Finding Optimal Bayesian Networks by Dynamic Programming*, Tech. Rep., Carnegie Mellon University, June 2005.

[85] P. Spirtes, C. Glymour, and R. Schemes, *Causation, Prediction, and Search*, 2 edition, The MIT Press, 2000.

[86] P. Sprites and C. Meek, "Learning Bayesian networks with discrete variables," *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, 1995, pp. 294 – 299.

[87] J. Suzuki, "Learning Bayesian Belief Networks Based on the Minimum Description Length Principle: Basic Properties," vol. E82-D, 1999, pp. 356–367.

[88] Y. Tamada, S. Imoto, and S. Miyano, "Parallel Algorithm for Learening Optimal Bayesian Network Structure," *Journal of Machine Learning Research*, vol. 12, 2011, pp. 2437–2459.

[89] M. Teyssier and D. Koller, "Ordering-Based Search: A Simple and Effective Algorithm for Learning Bayesian Networks," *Proceedings of the Twenty-First Conference Annual Conference on Uncertainty in Artif cial Intelligence (UAI-05)*, Arlington, Virginia, 2005, pp. 584–590, AUAI Press.

[90] J. Tian, "A Branch-and-Bound Algorithm for MDL Learning Bayesian Networks," *Proceedings of the 16th Conference on Uncertainty in Artif cial Intelligence*. 2000, pp. 580–588, Morgan Kaufmann Publishers Inc.

[91] I. Tsamardinos, L. Brown, and C. Aliferis, "The max-min hill-climbing Bayesian network structure learning algorithm," *Machine Learning*, vol. 65, 2006, pp. 31–78, 10.1007/s10994-006-6889-7.

[92] R. Valenzano, N. Sturtevant, J. Schaeffer, K. Buro, and A. Kishimoto, "Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agient Search Algorithms," *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 2010.

[93] T. Verma and J. Pearl, "Equivalence and synthesis of causal models," *Proceedings of the Sixth Annual Conference on Uncertainty in Artif cial Intelligence*, New York, NY, USA, 1991, UAI '90, pp. 255–270, Elsevier Science Inc.

[94] V. Vidal, L. Bordeaux, and Y. Hamadi, "Adaptive K-Parallel Best-First Search: A Simple but Eff cient Algorithm for Multi-Core Domain-Independent Planning," *International Symoposium on Combinatorial Search*, 2010.

[95] A. Wille and P. Bhlmann, "Low-order conditional independence graphs for inferring genetic networks.," *Statistical Applications in Genetics and Molecular Biology*, vol. 5, no. 1, 2006, p. Article1.

[96] P. J. Woolf, W. Prudhomme, L. Daheron, G. Q. Daley, and D. A. Lauffenburger, "Bayesian analysis of signaling networks governing embryonic stem cell fate decisions," *Bioinformatics*, vol. 21, no. 6, 2005, pp. 741–753.

[97] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe, "The K computer: Japanese next-generation supercomputer development project," *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, aug. 2011, pp. 371 –372.

[98] H. Yu, S. Zhu, B. Zhou, H. Xue, and J.-D. J. Han, "Inferring causal relationships among different histone modif cations and gene expression," vol. 18, 2008, pp. 1314–1324–.

[99] C. Yuan, H. Lim, and M. Littman, "Most Relevant Explanation: computational complexity and approximation methods," *Annals of Mathematics and Artif cial Intelligence*, vol. 61, 2011, pp. 159–183, 10.1007/s10472-011-9260-z.

[100] C. Yuan, H. Lim, and T.-C. Lu, "Most Relevant Explanation in Bayesian Networks," *Journal of Artif cial Intelligence Research*, vol. 42, 2011, pp. 309 – 352.

[101] C. Yuan and T.-C. Lu, "A general framework for generating multivariate explanations in Bayesian networks," *Proceedings of the 23rd national conference on Artif cial intelligence - Volume 2*. 2008, pp. 1119–1124, AAAI Press.

[102] C. Yuan, B. Malone, and X. Wu, "Learning Optimal Bayesian Networks using A* Search," *Proceedings of the 22nd International Joint Conference on Artif cial Intelligence*, 2011.

[103] W. Zhang and J. C. Pemberton, "Epsilon-transformation: exploiting phase transitions to solve combinatorial optimization problemsinitial results," *Proceedings of the twelfth national conference on Artif cial intelligence (vol. 2)*, Menlo Park, CA, USA, 1994, AAAI'94, pp. 895–900, American Association for Artif cial Intelligence.

[104] R. Zhou and E. Hansen, "Sweep A*: Space-eff cient heuristic search in partially ordered graphs," *Proceedings of 15th IEEE International Conf. on Tools with Artif cial Intelligence*, 2003, pp. 427–434.

[105] R. Zhou and E. A. Hansen, "Structured duplicate detection in external-memory graph search," *Proceedings of the 19th national conference on Artif cal intelligence*. 2004, AAAI'04, pp. 683–688, AAAI Press.

[106] R. Zhou and E. A. Hansen, "Breadth-f rst heuristic search," *Artif cial Intelligence*, vol. 170, 2006, pp. 385–408.

[107] R. Zhou and E. A. Hansen, "Parallel structured duplicate detection," *Proceedings of the 22nd national conference on Artif cial intelligence - Volume 2*. 2007, AAAI'07, pp. 1217–1223, AAAI Press.