EFFICIENT INFERENCE IN BAYESIAN NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF APPLIED PHYSICS
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Alexander V. Kozlov
March 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Daphne Koller
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jaswinder Pal Singh

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Walter A. Harrison

Approved for the University Committee on Graduate Studies:

Abstract

Bayesian networks are becoming an important tool for knowledge representation and reasoning under uncertainty. Bayesian networks allow intuitive causal representation of dependencies as well as efficient algorithms for probabilistic inference. Bayesian networks have been applied to problems in medical diagnosis, speech recognition, automated vision, machine learning, fault and yield diagnosis, scheduling and other areas. The number of practical applications is increasing rapidly as more and more areas encounter the need for decision making under uncertain circumstances.

A Bayesian belief network (BN) is a directed acyclic graph (DAG) whose nodes represent random variables and whose edges represent dependencies between these random variables. The dependencies are expressed as conditional probabilities of a node conditioned on the set of its parents. As a whole, a BN represents a joint probability distribution over the random variables in its domain, which is given by a product of conditional probabilities in the network.

A probabilistic inference in a Bayesian network is a computation which answers a quantitative question, called a query, about the probability of one event given information about some other events. For example, one might query the probability of a person having lung cancer given that he has a positive X-ray result. The inference task in a general BN is *NP*-hard and thus computationally expensive. This thesis addresses the problem of improving BN inference so that it scales up to much larger problems.

First, we show how parallelism can be exploited to speed up probabilistic inference. We find that exact probabilistic inference is extremely memory intensive as compared to more traditional scientific parallel applications, since the data set accessed grows

in proportion to the computation with a small constant. By utilizing special-purpose techniques for managing data locality, we manage to achieve almost linear speedup on medium scale machines (up to 64 processors) for real-world networks.

Parallelism by itself is insufficient for handling large and complex problems. In this case, the intermediate steps of the inference process generate probability distributions whose dimension is very large, rendering the inference process impractical. My thesis investigates the idea of doing approximate probabilistic inference by using more compact representations of these large distributions. We propose a new general approach for doing so, by abstracting (partitioning) the state space of a complex distribution. This thesis contains both techniques for choosing the abstraction and algorithms for probabilistic inference in the abstracted representation. We provide a technique to tailor the abstraction to the task at hand.

Our approach differs from existing approaches in that we choose to abstract the joint state space of a group of nodes, a clique in BN terms. This allows us to make the state space abstraction more effective and probabilistic inference in an abstracted network more efficient. We propose a practical algorithm to adjust the abstraction granularity during probabilistic inference based on the minimization of the information-theoretic measure of distance, the KL distance. Furthermore, we introduce a new hierarchical data structure for representing the clique state space abstraction, the Binary Split Partition (BSP) tree.

We demonstrate these techniques on a few examples. In BN2O networks (two-layer cause and effect networks), particularly useful for medical diagnosis tasks, we find a simple fixed static state space partitioning that reduces the computation time substantially at the expense of a small error for all possible diagnostic queries in the network. We extend the static abstraction to a dynamic abstraction in hybrid networks, networks with both discrete and continuous variables that are often useful in practice, and show how to choose the appropriate hierarchical state space partitioning dynamically based on the task and required precision. As a result, we provide the first effective general-purpose inference algorithm for hybrid networks.

Preface

"The only difference between me and a madman is that I am not mad."

Salvador Dali (1904-89)

This thesis is the result of my tumultuous graduate study at the department of Applied Physics at Stanford University from October 1991 to March 1998. During this time, I changed advisors three times, and was lucky to meet Prof. Daphne Koller who finally turned out to be my principal advisor. The topic of my dissertation, which finally turned out to be related to Artificial Intelligence, changed even a larger number of times.

Although I spent more time and effort than I might have wished to and may have preferred to work with Daphne Koller from the start, the experience of changing fields greatly enriched my knowledge and skills. This experience taught me to understand people with a different background talking a different scientific language. My training in other areas also helped me to obtain some non-traditional insights into the problem. I hope that the new way of looking at probabilistic inference described in this thesis will be useful for my colleagues.

Alexander V. Kozlov
Palo Alto, 1998

Acknowledgements

"... and to everyone who made it possible"

A cartoon caption showing a person standing on top of a human pyramid

I would like to acknowledge many people who contributed to the results of this thesis. First of all, I would like to thank my parents who had to put up with my absence during all these six long years. I am equally grateful to all my friends who encouraged me to come from Russia to study at Stanford University.

Looking backwards, I have to admit that I was lucky to come exactly to Stanford University with its tolerance to whatever interests a student might have and to whatever direction of research a student might undertake. My research here started with a series of Physics professors, which ended with a couple of years with Prof. Walter Harrison. It would have been impossible without him to understand what really interests me and to change my interests as drastically as from theoretical condensed matter physics to artificial intelligence. I am very thankful for his understanding and hope to maintain good relations with my physics mentors throughout my life.

I am also grateful to Jaswinder Pal Singh and John Hennessy who gave me a research assistantship and guidance to study parallelism in probabilistic inference. The results of this work and their effect on parallel architecture have yet to be studied more carefully, but I am grateful that it was I who was one of the first to parallelize an *NP*-hard problem on the most powerful computers today. Had they not undertaken the risk of taking a novice to program a parallel application it would have been impossible for me to work on this wonderful modern hardware and to propose my theory of “data intensity”. I hope that this idea will be useful for parallelizing other AI applications.

I have to thank Jaswinder Pal Singh separately. We spent many hours together just at the start of my Computer Science career, when I needed guidance and attention most. I always felt his genuine interest in my scientific progress, even after I stopped working with him directly.

I met my final advisor Prof. Daphne Koller when I was making my first attempt to graduate and, as a result, spent another year and a half at the Robotics Lab of the Computer Science department. I met many wonderful people there such as the Robotics Lab director Prof. Yoav Shoham, Prof. Nils Nilson, Nir Friedman, my officemate Xavier Boyen, my DAGS-mates Lise Getoor, Avi Pfeffer, Mehran Sahami, Uri Lerner, Eric Bauer, and others. I had many thought-provoking conversations with them and hope that I have left as much impact on this group as this group left on me.

I have to express special thanks to Daphne Koller whose enthusiasm in science and artificial intelligence served as an inspiration during all my years in the Robotics Lab. I apologize for all the pain I inflicted on her by my indifference and procrastination and appreciate her trust in me. Without her, my thesis would be much more shallow and much harder to understand. I appreciate her willingness to spend many hours with me and patience to reach some final understanding of the problems we were working on. I feel very sorry that I had to leave her and to finish my Ph.D. at this time. I am sure that another year or two might have been very productive in terms of new interesting results on probabilistic inference in general and on hierarchical reasoning in particular. Nevertheless, I am indebted to her at the same time because she allowed me to follow my inner voice to work on real practical problems and helped me to find an interesting position in data mining research in industry.

I have to express gratitude to many people in the UAI community who encouraged me to pursue the carrier in probabilistic reasoning. This includes Peter Cheeseman and Wray Buntine from NASA Ames who taught a course at the CS department at Stanford and were always curious about my progress, Greg Provan, Malcolm Pradhan, and Max Henrion from Stanford Medical School who helped me with the BN2O research, discussed their ideas and mine, and provided me with the proprietary CPCS network. I am thankful to R. Miller and R. Parker from Medical School of the University of Pittsburgh, the owner of the CPCS network, for the permission to use the CPCS BN in my research.

Finally, I have to say a lot of good things about the administration of the CS department and particularly about Jutta McCormick and Mina Madrigal of the Robotics Lab, which were very different from the administration of the Applied Physics department of Stanford University as well as the administration of Moscow State University. It was very inspiring and encouraging to have such open minded and helping administrators, a fact that I could hardly believe at first.

I apologize if I did not mention any of the other people who directly or indirectly helped me with this dissertation. At the very end, I would like to thank my parents once again since all that I am is just what my parents made me to be.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Inference is a process of obtaining new knowledge from existing knowledge. It is at the heart of all computer-based decision-making systems. In this thesis, I consider efficiency of inference in a graphical structure called a *Bayesian network* [Pearl, 1988; Heckerman and Wellman, 1995], which got its name from Bayes' rule for computing conditional probabilities of causally related random variables [Bayes, 1763]. Bayesian networks are also called *belief networks*, *causal probabilistic networks*, *directed Markov fields*, or, given some additional structure, *influence diagrams*. Bayesian network inference is based on the *theory of probability* and is often called *probabilistic inference.* Probabilistic inference has been used in a number of practical systems.

A Bayesian belief network (BN) consists of nodes, which represent random variables taking a set of values, and directed edges, which represent direct probabilistic dependencies between these variables. Each node is assigned a conditional probability for itself conditioned on its parents in the graph. More precisely, the graphical structure specifies a set of conditional independence statements, and the numerical values of the conditional probabilities characterize the strength of probabilistic dependencies [Pearl, 1988]. As a whole, a BN provides a compact representation for the joint probability distribution, the probability distribution over the joint state space of all variables in the network, by a product of all conditional probabilities associated with the nodes in the network.

Given a BN, we can answer a question, called a query, about the probability of a random variable taking one of its values, conditioned on the fact that we know the values of some other random variables. The process of inferring such a probability or a set of probabilities is called probabilistic inference. In general, an input to a probabilistic inference system based on a BN is *evidence* about the values of some variables(s), say the results of physical examination and tests performed on a hospital patient, and the output is a *probability distribution* over the values of some other variable(s), say the disease that the patient has; the BN in this case describes a set of *expert beliefs* about the patient and the disease in question. A BN representation of knowledge has many benefits: it allows intuitive causal interpretation, is elegant, concise, and modular.

Probabilistic inference in a BN is computationally intensive and is known to be *NP*-hard in general [Cooper, 1990]; the costs grow exponentially with the size of a BN. It is crucial to deal with this issue since the computational tasks in many practical applications involve hundreds and thousands of variables, making the cost prohibitively large even for modern computers.

In this thesis, I investigate two ways of dealing with the computational complexity of probabilistic inference. The first one is to apply parallel processing, where we partition the computation workload between several processors in an attempt to reduce computation time of exact probabilistic inference. The other one is to make approximations in the probabilistic inference, where we try to get the best estimate of the output probabilities in a limited amount of computation time.

Let us begin by formally defining a BN and looking at some simple examples of BNs and probabilistic inference.

1.1 BN definition & examples

A BN is a joint probability distribution over random variables represented as a product of conditional probabilities. The random variables in a BN are formally represented by nodes. The probability of each node is determined by the probability distribution for the values of the node parents in the BN graph. This dependency is formally represented by a conditional probability $p(x_i|Pa(x_i))$ associated with the node, where

x_i denotes a random variable and $Pa(x_i)$ denotes the set of node parents.

Thus, the joint probability distribution of over *all* random variables in a BN is formally given by the chain rule:

$$p(x_1, \ldots x_n) = \prod_{i=1}^{n} p(x_i | Pa(x_i)). \tag{1.1}$$

A joint probability entry formally gives the probability of an event with a fixed set of values for all variables $x_1, \ldots, x_n$ in the network.

For example, let us consider the "Chest Clinique" expert knowledge from a medical domain [Lauritzen and Spiegelhalter, 1988]. Tuberculosis and lung cancer create an abnormality in the chest. Either of these abnormalities can cause positive chest X-ray and dyspnea (shortness of breath). Another probable cause of dyspnea is bronchitis. Tuberculosis is more likely in people who have visited Asia. Lung cancer and bronchitis, in their turn, are more often encountered in smokers. These facts are incorporated in the graphical structure of a "Chest Clinique" BN shown in Fig. 1.1.

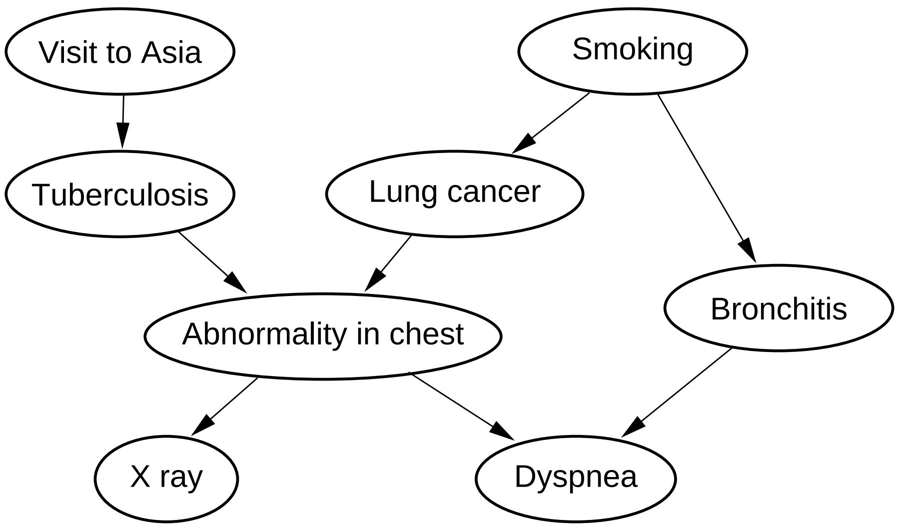


Figure 1.1: The "Chest Clinique" BN describing statistical dependencies between predisposing factors, diseases, and symptoms of a chest clinic patient.

A node in Fig. 1.1 represents a random variable. Although a random variable in general can have any number of values, we assume that each of the variables in the "Chest Clinique" BN can have only two possible values: either *true* or *false*. For example, if the patient has visited Asia, we say that the value of the variable x_V corresponding to the node "Visit to Asia" is *true*. On the other hand, if the patient has not visited Asia, we say that the value of this variable is *false*. The facts about the patient can be found from the patient data sheet and tests and are called *findings* in medical literature.

Similarly, the fact that the patient has or has not the disease tuberculosis is reflected by the *true* or *false* value of the variable x_T corresponding to the disease node "Tuberculosis". Although the value of this variable might not be available prior to the final diagnosis, it is possible to have beliefs about it. The probability of the variable x_T to be in either state might affect the treatment of the patient.

An edge in Fig. 1.1 represents a direct dependence. For example, we believe that the probability of lung cancer is likely to be affected by smoking. On the other hand, there is no apparent correlation between smoking and the probability of visiting Asia, and we do not have an edge between nodes "Smoking" and "Visit to Asia". We say that the random variables corresponding to the nodes "Smoking" and "Visit to Asia" are independent.

Although there is no edge between nodes "Visit to Asia" and "Abnormality in chest", the corresponding to these nodes variables are dependent through the node "Tuberculosis". Chest abnormalities are more likely in people who visited Asia since they are more likely to have tuberculosis. Thus, the events "Visit to Asia" and "Abnormality in chest" can affect each other.

However, if the value of variable x_T is given, i.e., we know that the patient has or has not tuberculosis, the variables x_A and x_V become independent. A visit to Asia does not directly affect the probability of a person having an abnormality in lungs. We say that the variables x_A and x_V are conditionally independent given the variable x_T. In general, a node is conditionally independent of all non-descendants given its parents.

A conditional probability associated with each node in the graph is used to quantify the dependencies in a BN. For example, we associate a conditional probability $p(x_D|x_A, x_B)$ of the variable x_D corresponding to the node "Dyspnea" conditioned on the variables x_A and x_B corresponding to the nodes "Abnormality in chest" and "Bronchitis". The values for this conditional probability might be given by Table 1.1 for instance, as they are in the original formulation of this network [Lauritzen and Spiegelhalter, 1988].

x_A	x_B	$p(x_D = true)$
false	*false*	0.1
false	*true*	0.8
true	*false*	0.7
true	*true*	0.9

Table 1.1: Values for the conditional probability $p(x_D|x_A, x_B)$ of the node x_D, "Dyspnea", conditioned on the nodes x_A, "Abnormality in chest", and x_B, "Bronchitis". All these random variables are assumed to have only two possible values *false* and *true*.

The "Chest Clinique" BN defines a *joint probability distribution* over all variables x_V, x_T, x_A, x_X, x_L, x_S, x_B, and x_D in the network. For this network, the full joint probability distribution table has $2^8 = 256$ entries. All these entries are represented compactly by the product of all conditional probabilities in the "Chest Clinique" BN:

$$\begin{aligned} p(x_V, \ldots, x_D) = {} & p(x_D|x_A, x_B)p(x_B|x_S)p(x_S) \\ & p(x_X|x_A)p(x_A|x_T, x_L)p(x_L|x_S)p(x_T|x_V)p(x_V). \end{aligned} \tag{1.2}$$

The joint probability distribution defined by a BN is used for probabilistic inference.

The set of values for a random variable in a BN can also be continuous. A BN in this case is called a *hybrid BN* since discrete and continuous variables are usually intermixed. For instance, the BN shown in Fig. 1.2 describes a statistical dependence of the average daily temperature on the season of the year. The "Temperature"

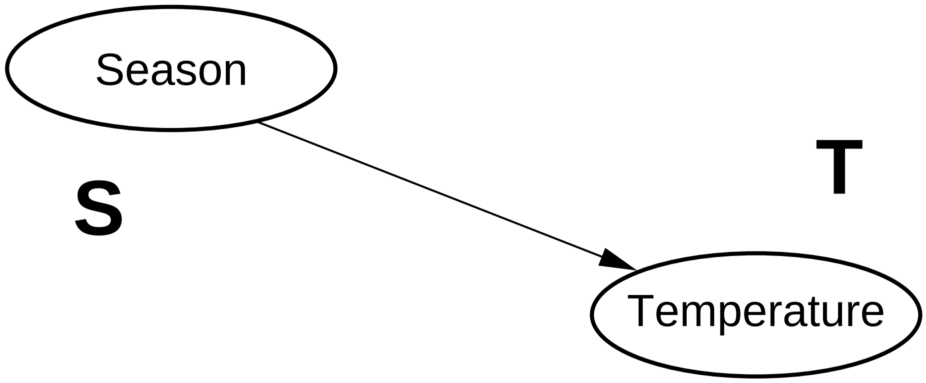


Figure 1.2: "Temperature" Bayesian network, which is a hybrid BN and describes a statistical dependence of the average daily temperature on the year season.

node represents a continuous variable x_T, and the "Season" node represents a four-valued discrete variable x_S which can take one of the four values: *"Winter"*, *"Spring"*, *"Summer"*, or *"Autumn"*.

The edge in Fig. 1.2 represents a dependence between the continuous random variable "Temperature" and the discrete random variable "Season", which is quantified by a *probability density function*. For example, we might specify the dependence between variable x_T and x_S by a normal distribution:

$$N(x_T; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-(x-\mu)^2/2\sigma^2) \tag{1.3}$$

with the average temperature $\langle \mathrm{T} \rangle = \mu$ and variance $\langle \mathrm{T}^2 \rangle - \mu^2 = \sigma^2$, both of which depend on the discrete value of the x_S variable. In this representation, the dependence between x_T (temperature) and x_S (season) might be given by Table 1.1.[1]

One of the questions we can ask the "Temperature" network is what the probability of the season being "Winter" is if the current average daily temperature is

[1]The data are for Berkeley, California, 1989; the data were taken from The Earth System Science Community web-page at `http://www.circles.org`.

x_S	μ	σ
"Winter"	9.4	2.6
"Spring"	14.7	7.6
"Summer"	17.3	1.6
"Autumn"	15.3	6.6

Table 1.2: Values for the dependence $p(x_T|x_S)$ of the node x_T, "Temperature", conditioned on the nodes x_S, "Season". The temperature has a normal distribution whose parameters depend on the season of the year.

12°C, thus modeling an intuitive human perception of a season: the low temperature corresponds to the winter and the higher temperature corresponds to the summer.

Of course, the values in Table 1.1 can describe the season-temperature dependence only for one particular region, which was Berkeley in the above case. However, both the "Chest Clinique" and the "Temperature" BNs can be refined by adding additional nodes and details and made more general and useful. For example, in the "Temperature" BN we can include the dependence of the average daily temperature on such factors as shade, altitude, distance from the Pacific Ocean, all of which can also be expressed as random variables. However, we will see that the increase in the number of variables and dependencies between variables makes probabilistic inference much harder computationally. In general, probabilistic inference time increases by a factor for each additional variable. Thus, it grows exponentially with the number of variables in a general BN.

1.2 Existing applications

Above we have shown two very simple BNs. Let us look at the practical examples of BNs which are used in practice.

Disease diagnosis is a traditional application for BNs. A typical query in a diagnostic BN is about the probability of a disease given a set of findings. One of the first medical BNs was built by David Heckerman at Stanford Medical School for practical lymph node disease diagnosis and was called PATHFINDER [Heckerman, 1990; Heckerman *et al.*, 1992]. It contains several simplifying assumptions built in, for example a *single fault* hypothesis, which significantly reduces the complexity of the network construction process and the probabilistic inference computation.

The two largest practical BNs also come from a medical project called QMR (Quick Medical Reference, [Shwe *et al.*, 1991]). The QMR-DT (Decision-Theoretic Quick Medical Reference) BN is a two layer medical diagnostic network with a simplified interaction between nodes, which facilitates probabilistic inference (see a detailed description of BN2O networks in Chapter 5). The QMR-DT BN has over 600 disease nodes, 4000 finding nodes, and 40,000 disease-finding links. The other BN in the QMR project, the CPCS (Computer-based Patient Case Study) BN for internal disease diagnosis (see Fig. 1.3), has 448 nodes and 908 arcs and has multiple layers. Techniques for efficient inference in large networks such as these was one of the main goals of this dissertation.

Probabilistic encoding of information has been used for time critical applications, making efficient inference crucial. For example, the Vista system [Horvitz and Barry, 1995] is a decision-theoretic system that has been used at NASA Mission Control Center in Houston. The system uses Bayesian networks to interpret live telemetry and provides advice on the likelihood of alternative failures of the space shuttle's propulsion systems; it considers time criticality and recommends actions of the highest expected utility. The Vista system employs decision-theoretic probabilistic methods for controlling the display of information to dynamically identify the most important information to highlight.

Belief networks in one form or another have been used for fault diagnosis (Windows '95, Intel processor fault diagnosis, airplane engines), robot control (optimizing actions under uncertainty), real-time monitoring (freeway traffic surveillance), forecasting (weather, stock market, oil prices, crop production), speech recognition, image analysis, and information retrieval. The number of applications is increasing rapidly.[2]

[2]See the UAI web sites devoted to BN applications `http://www.auai.org/BN-Routine.html`.

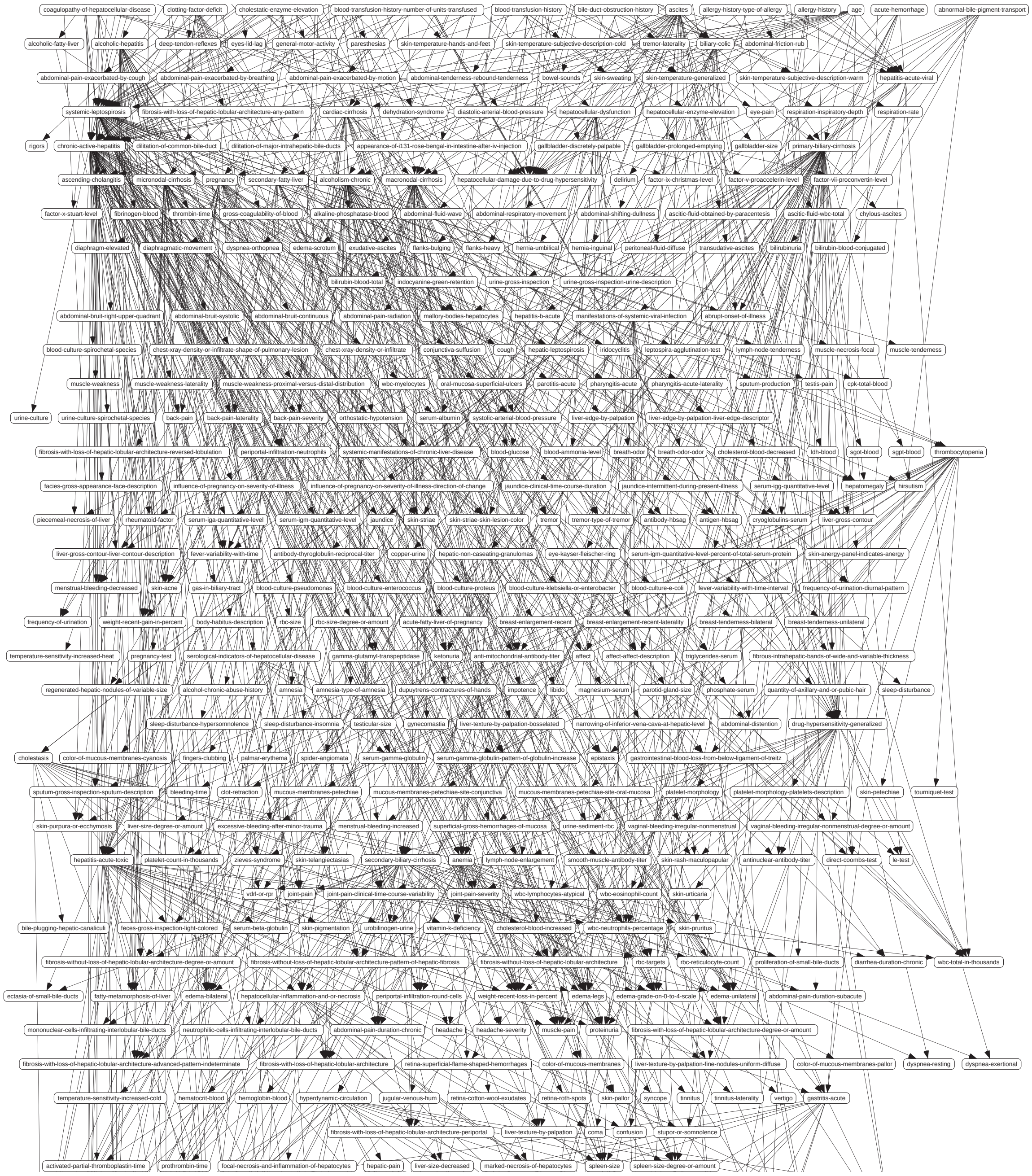


Figure 1.3: A fragment of the CPCS (Computer-based Patient Case Simulation) medical diagnostic network. The full network contains 448 nodes and 908 arcs.

1.3 Probabilistic inference

Probabilistic inference answers a question about the probability distribution over the values of *query* variables given some other *evidence* variables. For example, in a medical application we compute the probability distribution over possible diseases given the findings. In the "Chest Clinique" BN, a query might be to compute $p(x_L = \mathit{true}|x_X = \mathit{true}, x_S = \mathit{false})$, i.e., to compute the probability of a patient having cancer if he has positive X-ray results and does not smoke. In the "Temperature" network, a query might be $p(x_S = \text{"}\mathit{Winter}\text{"}|x_T = 12°\mathrm{C})$, i.e., to compute the probability of winter if the daily average temperature outside is 12°C.

Probabilistic inference can be represented as an application of the Bayes' rule to the joint probability distribution defined by a BN. For example, to compute the probability $p(x_S = \text{"}\mathit{Winter}\text{"}|x_T = 12°\mathrm{C})$, we notice that it is the ratio of two probabilities:

$$p(x_S = \text{"}\mathit{Winter}\text{"}|x_T = 12°\mathrm{C}) = \frac{p(x_S = \text{"}\mathit{Winter}\text{"}, x_T = 12°\mathrm{C})}{p(x_T = 12°\mathrm{C})}. \tag{1.4}$$

Each of the probabilities on the right side might be computed from the joint probability distribution of the "Temperature" BN:

$$\begin{aligned}
&p(x_S = \text{"}\mathit{Winter}\text{"}|x_T = 12°\mathrm{C}) \\
&= \frac{p(x_S = \text{"}\mathit{Winter}\text{"}, x_T = 12°\mathrm{C})}{\begin{array}{r} p(x_S = \text{"}\mathit{Winter}\text{"}, x_T = 12°\mathrm{C}) + p(x_S = \text{"}\mathit{Spring}\text{"}, x_T = 12°\mathrm{C}) + \\ p(x_S = \text{"}\mathit{Summer}\text{"}, x_T = 12°\mathrm{C}) + p(x_S = \text{"}\mathit{Autumn}\text{"}, x_T = 12°\mathrm{C}) \end{array}} \\
&= \frac{N(12; 9.4, 2.6)}{N(12; 9.4, 2.6) + N(12; 14.7, 7.6) + N(12; 17.3, 1.6) + N(12; 15.3, 6.6)} \approx 0.47,
\end{aligned}$$

while the probability of summer computed by the same algorithm is only 0.005.

Probabilistic inference becomes more involved in multiply connected networks, networks where we can have multiple paths from one node to another. In the latter case, we have to evaluate sums over large state spaces of groups of variables. In general, computation time and memory requirements grow exponentially with the size of a general BN. Since more general and more useful BNs contain many variables, it is important to develop techniques to curb the computational complexity of probabilistic inference.

1.4 Overview of the thesis

Probabilistic inference is practically important and computationally intensive. Thus, we need a better understanding and more efficient implementations of probabilistic inference. We address these problems from two different directions: parallel processing and approximation of probabilistic inference, which constitute the two parts of the thesis. The first part (Chapters 2 and 3) deals with data intensity and parallelization of probabilistic inference algorithm. The second part (Chapters 4 to 6) deals with approximations in probabilistic inference that allow reduction of computation time at the expense of small errors in the results.

Chapter 2 is a more detailed introduction to probabilistic inference. We take an optimal factoring approach to describing probabilistic inference and show how inference in BNs can be reduced to the summation of the joint probability distribution. The algorithm analysis conducted in this chapter helps us to understand the structure of computations and data dependencies. In particular, it helps to understand why probabilistic inference is a very data intensive application. The amount of memory and the amount of computations grow at the same rate and the average number of processor operations per byte of memory is small.

In Chapter 3 we discuss multiprocessor implementations of probabilistic inference. For a successful parallel implementation, we need to ensure load balance—an even distribution of the workload between the processors—and data locality—reuse of the processor's cached data, as to avoid unnecessary interprocessor communication and reduce memory system time. It is often very difficult to find an optimal tradeoff between load balance and data locality [Stenström and Dahlgren, 1996]. As we show in Chapter 3, probabilistic inference is a particularly interesting application from the point of view of this tradeoff.

Our multiprocessor implementation achieves an almost linear speedup with the number of processors. However, we can achieve significantly greater speedup by making approximations during probabilistic inference. To achieve the reduction in memory requirements, inference time, and a better speedup we need to reason on a new abstract level. We propose a new general mechanism for abstracting parts of the problem. It approximates the details of the problem by concentrating on its most important parts. Although approximating probabilistic inference is still *NP*-hard [Dagum and Luby, 1993], we can obtain a substantial computation time reduction due to proper approximations. This approach is discussed in detail in Chapter 4.

In the next two chapters we apply the above abstraction idea for two different problems. We start with a static abstraction partitioning in Chapter 5 and show that we can get substantial computation time savings at the expense of a small relative error of the result in BN2O networks. In Chapter 6, we show a technique of choosing proper abstraction dynamically by an iterative algorithm. We apply this technique to hybrid networks containing continuous variables. As a result, we obtain the first general purpose algorithm for hybrid networks with arbitrary dependence between continuous and discrete variables.

Finally, in Chapter 7, we summarize our results and consider possible extensions of the current work.

1.5 Contributions

The major contributions of the first part are:

- We show that data locality is extremely important for the efficiency of probabilistic inference and show how to manage the data locality in general probabilistic inference and for specific networks on commercially available computer systems.

- We provide two multiprocessor implementations of probabilistic inference. The first one exploits only one type of concurrency and the second tries to use all of the available concurrency but compromises data locality. We analyze the performance of the two implementations and show that the first one outperforms the second for practical networks due to better data locality.

- We show that there is enough concurrency in practical networks for medium scale parallel computers. The best implementation achieves close to ideal speedup for the CPCS medical diagnostic network.

The major contributions of the second part are:

- We develop a general approach to approximating probabilistic inference based on the relative entropy or Kullback-Leibler (KL) distance and state space abstraction paradigm.

- We demonstrate how to statically apply the state space abstraction on the example of BN2O networks and obtain a substantial computation time reductions at the expense of a small relative error of the result.

- We demonstrate a general approach for dynamic state space abstraction based on our notion of WKL distance and an iterative algorithm.

- We develop a general purpose algorithm for hybrid networks with arbitrary dependence between continuous and discrete variables nodes in arbitrary topology and demonstrate its performance.

Chapter 2

Probabilistic inference algorithms

"Now the rest of the acts of Basha and what he did and his might, are they not written in the Book of the Chronicles of the Kings of Israel?"

1 Kings 16:5

A probabilistic inference computation is a computation that finds a conditional probability of the *query* random variable conditioned on the given *evidence* random variable. In this chapter, we first analyze this computation from the *optimal factoring approach* point of view first proposed by D'Ambrosio [Li and D'Ambrosio, 1994], then compare it to the more traditional *Lauritzen-Spiegelhalter approach* which historically appeared first [Jensen *et al.*, 1990; Lauritzen and Spiegelhalter, 1988]. Finally, we consider a spectrum of approaches to managing computational complexity by making approximations in inference.

2.1 Probabilistic inference

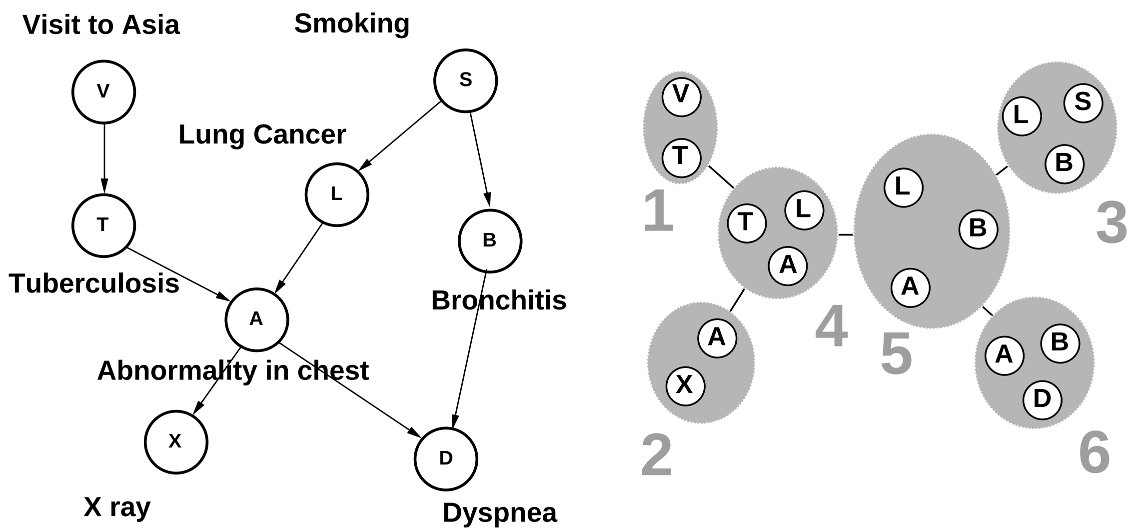
Probabilistic inference can be reduced to evaluation of marginals over some variables. Let us see how we can perform these computations efficiently on the example of "Chest Clinique" BN.

2.1.1 Optimal factoring

A conditional probability, which constitutes a probabilistic query, can be expressed as a ratio of two probabilities. For example, the conditional probability that a patient has cancer ($x_L = true$) given that he has a positive X-ray result ($x_X = true$) and does not smoke ($x_S = false$) can be expressed as the ratio:

$$p(x_L = true | x_X = true, x_S = false) = \frac{p(x_L = true, x_X = true, x_S = false)}{p(x_X = true, x_S = false)}, \quad (2.1)$$

where $p(x_L = true, x_X = true, x_S = false)$ is the probability that all three events specified in the query are happening together and $p(x_X = true, x_S = false)$ is the probability of the findings only. Let us look at the "Chest Clinique" BN once again.



(a) "Chest Clinique" BN (b) Join tree

Figure 2.1: The structure of the "Chest Clinique" BN also shown in Fig. 1.1 and one of the possible join trees used to compute queries in the text. All conditional probabilities in the original network, which are functions of the states of a node and its parents, are wholly contained within one of the cliques in the join tree.

If we denote the nodes in the "Chest Clinique" BN by x_V, x_T, x_A, x_X, x_L, x_S, x_B, and x_D as we did in the introduction, the above probabilities can be computed as sums of the joint probability distribution (1.2). The probability $p(x_L = \mathit{true}, x_X = \mathit{true}, x_S = \mathit{false})$ is:

$$p(x_L = \mathit{true}, x_X = \mathit{true}, x_S = \mathit{false}) = \sum_{\{x_V, x_T, x_A, x_B, x_D\}} p(x_V, x_T, x_S = \mathit{false}, x_L = \mathit{true}, x_A, x_X = \mathit{true}, x_B, x_D), \quad (2.2)$$

i.e., the sum of the joint probability distribution over variables x_V, x_T, x_A, x_B and x_D, and the probability $p(x_X = \mathit{true}, x_S = \mathit{false})$ is:

$$p(x_X = \mathit{true}, x_S = \mathit{false}) = \sum_{\{x_V, x_T, x_L, x_A, x_B, x_D\}} p(x_V, x_T, x_S = \mathit{false}, x_L, x_A, x_X = \mathit{true}, x_B, x_D), \quad (2.3)$$

i.e., the sum of the joint probability distribution over variables x_V, x_T, x_L, x_A, x_B and x_D. In both cases we sum over all variables which do not appear in the left hand side.

Thus, an evaluation of a query is reduced to the summation of the joint probability distribution. Given the complete table for the "Chest Clinique" joint probability distribution, we need to sum $2^5 = 32$ terms to evaluate (2.2) and $2^6 = 64$ terms to evaluate (2.3). The summations can be carried out more efficiently by exploiting the structure of the joint probability decomposition (1.2). Let us see how it can be done.

We start with a simple computation of the probability $p(D) = p(x_D = \mathit{true})$ to demonstrate our ideas:

$$p(D) = \sum_{\{x_V, x_T, x_A, x_X, x_L, x_S, x_B\}} p(x_V, \ldots, x_A, x_D = \mathit{true}) =$$
$$\sum_{\{x_V, x_T, x_A, x_X, x_L, x_S, x_B\}} p(x_D = \mathit{true}|x_A, x_B) p(x_B|x_S) p(x_S)$$
$$p(x_X|x_A) p(x_A|x_T, x_L) p(x_L|x_S) p(x_T|x_V) p(x_V) \quad (2.4)$$

which requires $2^7 - 1 = 127$ summations. However, we might notice that the variable

x_V appears only in the two last terms: in the conditional probability $p(x_T|x_V)$ of the node x_T conditioned on the node x_V and in the probability $p(x_V)$ of the node x_V itself. None of the other terms contains the variable x_V. Thus, we can take all other terms as a common multiplier out of the summation over x_V and represent the sum (2.4) as:

$$p(D) = \sum_{\{x_V, x_T, x_A, x_X, x_L, x_S, x_B\}} p(x_V, \ldots, x_A, x_D = \mathit{true}) =$$
$$\sum_{\{x_T, x_A, x_X, x_L, x_S, x_B\}} p(x_D = \mathit{true}|x_A, x_B) p(x_B|x_S) p(x_S)$$
$$p(x_X|x_A) p(x_A|x_T, x_L) p(x_L|x_S) \sum_{x_V} p(x_T|x_V) p(x_V) \quad (2.5)$$

which requires two summations for the sum over x_V and $2^6 - 1 = 63$ summations for the sum over $\{x_T, x_A, x_X, x_L, x_S, x_B\}$. Thus, it will take only 65 summations to compute the same result as in (2.4) if we take $p(x_T|x_V)p(x_V)$ out as a common multiplier. The number of multiplications decreases also: (2.4) requires $2^7 \times 7 = 896$ and (2.5) requires $2^2 + 2^6 \times 6 = 388$ multiplications, which is more than a factor of two speedup.

We say that the product $p(x_T|x_V)p(x_V)$ constitutes the *clique potential* of the *clique* $C_1 = \{x_T, x_V\}$ (see Fig. 2.1(b)).[1] More generally, we say that a clique potential is a factor in the decomposition defined by one of the possible product decompositions, and a clique is a subset of nodes of the original network given by the decomposition.

A clique potential summed over corresponding variables constitutes a *message*. More generally, a message is a partial result during probabilistic inference computation. For example, the clique C_1 sends a message, which is C_1 potential summed over the variable x_V, to the rest of the network. This message contains all information the rest of the network has to know about this part of the BN for probabilistic

[1] The term *clique* appeared for historical reasons. The groups of variables were obtained by forming maximal cliques in a moralized and triangulated BN graph [Lauritzen and Spiegelhalter, 1988]. The cliques by themselves form a *clique tree* or a *join tree*. The algorithms to obtain the join tree from the network structure are well described in [Neapolitan, 1990].

inference—as we can see, the factors $p(x_T|x_V)$ and $p(x_V)$ enter nowhere else in the computation.

In other words, probabilistic inference can be viewed as local clique computations and message passing in a graphical structure called a *join tree*. One of the possible join trees, one that corresponds to the decomposition:

$$p(D) = \sum_{\{x_A, x_B\}} [p(x_D = true|x_A, x_B)] \qquad (C_6) \tag{2.6}$$

$$\sum_{x_L} [1] \qquad (C_5) \tag{2.7}$$

$$\sum_{x_T} [p(x_A|x_T, x_L)] \qquad (C_4) \tag{2.8}$$

$$\sum_{x_S} [p(x_B|x_S)p(x_L|x_S)p(x_S)] \qquad (C_3) \tag{2.9}$$

$$\sum_{x_X} [p(x_X|x_A)] \qquad (C_2) \tag{2.10}$$

$$\sum_{x_V} [p(x_T|x_V)p(x_V)]\,, \qquad (C_1) \tag{2.11}$$

is shown in Fig. 2.1(b) (the clique numbers on the right correspond to the clique numbers in the figure).[2] The above evaluation of $p(x_D = true)$ takes 19 summations and 56 multiplications as opposed to 127 summations and 896 multiplications in the original evaluation (2.4), producing a speedup of over 10. In general, the decomposition leads to substantial savings in sparse networks where variables appear only in a few factors.

Alternatively, one can represent the computation (2.6 – 2.11) as propagation of messages along the join tree edges up to the root, as shown in Fig. 2.2. Each clique multiplies the incoming message(s) with the assigned conditional probabilities, if any, and computes an outgoing message by the summation. Thus, the computation in (2.6 – 2.11) corresponds to propagating messages in the order $C_1 \to C_4$ (2.11), $C_2 \to C_4$ (2.10), $C_3 \to C_5$ (2.9), $C_4 \to C_5$ (2.8), $C_5 \to C_6$ (2.7).

[2] Note that the clique C_5 does not have assigned conditional probabilities and its potential is identically one.

Let us look how these computations can be mapped to the join tree computations. The propagation starts at a *leaf* (C_1, C_2, or C_3 in our example). A leaf clique computes its potential by multiplying the conditional probabilities assigned to it and forms a message to the parent clique. An *intermediate clique* (C_4 or C_5 in our example) receives the messages from the children, computes its potential by multiplying the assigned conditional probabilities and the messages, and forms a message to its parent. The propagation stops at the *root* (C_6 in our example) which contains the desired answer to a query (see (2.6)).

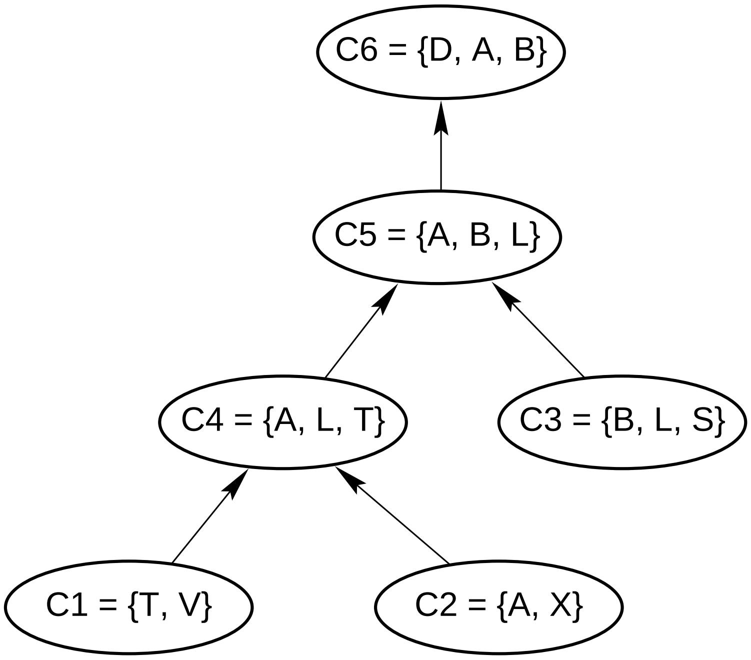


Figure 2.2: The clique propagation tree for the evaluation of probability of dyspnea $p(D)$ in the "Chest Clinique" BN. Messages are propagated from the leaves to the root of the tree.

In certain cases, the computation can be simplified further. For example, we might notice that the sum in the clique C_2 is identically one:

$$\sum_{x_X} p(x_X|x_A) \equiv 1 \tag{2.12}$$

due to the basic properties of conditional probabilities. Thus, the clique C_2 can be completely eliminated from the propagation tree,[3] and one needs only 17 summations and 56 multiplications to compute (2.4).

The savings also increase as the number of instantiated nodes increases. For example, the evaluation of (2.2) can be decomposed as:

$$p(x_L = \mathit{true}, x_X = \mathit{true}, x_S = \mathit{false}) =$$

$$\sum_{x_A, x_T} p(x_A | x_T, x_L = \mathit{true}) p(x_L = \mathit{true} | x_S = \mathit{false}) p(x_S = \mathit{false}) p(x_X = \mathit{true} | x_A) \sum_{x_V} p(x_T | x_V) p(x_V), \quad (2.13)$$

where we have to sum only over 3 variables x_A, x_T, and x_V and need only 5 summations and 20 multiplications. The above decomposition is equivalent to message propagation in the following order $C_1 \to C_4$, $C_2 \to C_4$, $C_5 \to C_4$. To evaluate (2.3), we need to sum (2.13) over x_L, which results in four additional summations.

The query (2.1) computations, $p(x_L = \mathit{true}, x_X = \mathit{true}, x_S = \mathit{false})$ (2.2) and $p(x_L = \mathit{true}, x_X = \mathit{true}, x_S = \mathit{false})$ (2.3), can be efficiently combined in one join tree propagation. For instance, the computations in the C_1 clique are completely identical for the two propagations and differ only at the clique C_4. In other words, the results of local computations can be cached and reused, resulting in further speedup compared to direct evaluation of the joint probability sums.

2.1.2 LS algorithm

Let us look at the probabilistic inference from another prospective. The above optimal factoring algorithm constitutes only a part of the initialization step, the *up* propagation, of a more general Lauritzen-Spiegelhalter (LS) algorithm [Jensen *et al.*, 1990]. The data structures in the LS algorithm might be reused for multiple queries. Let us see how it can be done.

In the LS algorithm, we also build a join tree. Each clique in the join tree contains

[3] This reduction corresponds to barren node reduction used for qualitative probabilistic inference in [Shachter, 1986].

a data structure with a *potential*, which we denote $\Psi(C_k)$ here, and each edge $\langle l, m\rangle$ in the join tree contains a data structure with a *message* between cliques C_l and C_m, which we denote $\Phi(C_l \cap C_m)$ here, defined on the intersection $C_l \cap C_m$ of the cliques C_l and C_m.

Each conditional probability $p(x_i|Pa(x_i))$ in the original network of a node x_i conditioned on its parents $Pa(x_i)$ is assigned to a clique which wholly contains the union $x_i \cup Pa(x_i)$ of the node x_i itself and its parents $Pa(x_i)$. The clique potentials $\Psi(C_k)$ in the cliques are initialized to the product of the conditional probabilities that are assigned to cliques. The messages between cliques are initialized to one, i.e., $\Phi(C_l \cap C_m) = 1$, so that the following property holds:

$$p(x_1, x_2, \ldots, x_n) = \frac{\prod_{\text{all potentials}} \Psi(C_k)}{\prod_{\text{all messages}} \Phi(C_l \cap C_m)}. \tag{2.14}$$

The above property is just another form of the joint probability decomposition.

The potentials and messages are called *self-consistent* if for each clique C_l and each adjacent to the clique edge $\langle l, m\rangle$ the following equality holds:

$$\sum_{x_i \in C_l \setminus (C_l \cap C_m)} \Psi(C_l) = \Phi(C_l \cap C_m). \tag{2.15}$$

In this case, the marginals for any variable can be obtained from any clique as if they were obtained from the whole joint probability distribution. For instance, if we want to find the probability $p(x_i)$ of a node x_i and $x_i \in C_l$, then:

$$p(x_i) = \sum_{x_j \in C_l \setminus x_i} \Psi(C_l). \tag{2.16}$$

The marginals obtained from any potential $\Psi(C_k)$ or message $\Phi(C_l \cap C_m)$ are guaranteed to be consistent. It means that to obtain the posterior distribution over a variable, we do not need to sum the whole joint probability distribution, but only potentials of one clique with a much smaller state space.

The problem is that making the potentials and messages in the join tree self-consistent is equivalent to doing as much work as in the optimal factoring approach.

To make the potentials and messages self-consistent is the major purpose of the *initialization* step which consists of the *up* propagation up the join tree to the root and the *down* propagation from the root to the leaves.

```
build a join tree for the BN graph
assign conditional probabilities to cliques
choose one of the cliques to be the root
for each clique starting from the leaves and up to the root do
    form clique potential by multiplying assigned conditional probabilities
    multiply the potential by the messages from descendants, if any
    form a message to the parent, if any, by summing over variables which do not
    appear in the parent
end for
```

Figure 2.3: The up propagation algorithm.

The pseudo-code for the up propagation is shown in Fig. 2.3 and is identical to the optimal factoring computation except that we store the potential and message arrays in the data structures. The result of the up propagation is the correct probability distribution in the root clique.

```
for each clique starting from the root down to the leaves do
    calibrate the potential by the message from the ancestor, if any
    form messages to the children, if any, by summing over variables which do not
    appear in a child
end for
```

Figure 2.4: The down propagation algorithm.

To get the correct probability distribution in the rest of the cliques, we need to propagate the missing information to the rest of the cliques in the tree. This is the purpose of the down propagation, the pseudo-code for which is shown in Fig. 2.4. The update rules for the down propagation are slightly different since the clique potential have already incorporated all the information below the given clique.

The clique potential update operation on the down the tree propagation stage is often called *calibration*. It differs from the optimal factoring or the up propagation in the LS algorithm by an extra division of the incoming message by the stored message on the edge. The intrinsic reason for this is that the message coming from the parent contains information about the whole tree, and we do not want to count the information stored in the message twice.

For example, let us assume that the C_4 clique needs to be calibrated by the C_5 clique (see Fig. 2.2). The new C_4 clique potential $\Psi'(x_A, x_L, x_T)$ is obtained from the old C_4 clique potential $\Psi(x_A, x_L, x_T)$, the stored old $\langle 4, 5\rangle$ message $\Phi(x_A, x_L)$, and the new $\langle 4, 5\rangle$ message $\Phi'(x_A, x_L)$ by:

$$\Psi'(x_A, x_L, x_T) = \frac{\Psi(x_A, x_L, x_T)}{\Phi(x_A, x_L)} \times \Phi'(x_A, x_L), \tag{2.17}$$

where $\Phi'(x_A, x_L)$ is computed in the clique C_5 by summation:

$$\Phi'(x_A, x_L) = \sum_{x_B} \Psi'(x_A, x_B, x_L). \tag{2.18}$$

The division is the "correction" not to take information from cliques C_1, C_2 and C_4 into account twice while accounting for the information coming from clique C_3, C_5 and C_6. After a complete update of the tree by the calibration operations the tree becomes self-consistent (the new stored message on the edge $\langle 4, 5\rangle$ is $\Phi'(x_A, x_L)$).

After the whole tree is calibrated, we can introduce *incremental* evidence into any of the cliques. To obtain posterior probability with this new evidence, we need to calibrate the tree once again by propagating messages from the instantiated clique to the rest of the network. For example, if we introduce evidence in the clique C_4, we will need to execute calibration in the order $C_4 \to C_1$, $C_4 \to C_2$, $C_4 \to C_5$, $C_5 \to C_3$, $C_5 \to C_6$.

From the Bayesian point of view, the calibration can be viewed as an application of the conditional probability rule (2.1) to the clique potentials. For example, if we introduce evidence to the clique $C_4 = \{x_A, x_L, x_T\}$, the new C_5 probability distribution $p'(x_A, x_B, x_L)$ is obtained by:

$$p'(x_A, x_B, x_L) = p(x_B|x_A, x_L)p'(x_A, x_L). \tag{2.19}$$

Given that the clique C_5 potential represented a consistent probability distribution, the conditional probability $p(x_B|x_A, x_L)$ is readily available and can be computed from the C_5 potential and $\langle 4, 5\rangle$ message:

$$p(x_B|x_A, x_L) = \frac{p(x_A, x_B, x_L)}{p(x_A, x_L)} = \frac{\Psi(x_A, x_B, x_L)}{\Phi(x_A, x_L)} \tag{2.20}$$

as well as the new probability $p'(x_A, x_L)$, which can be obtained from the C_4 potential updated by the new evidence:

$$p'(x_A, x_L) = \sum_{x_T} p'(x_A, x_L, x_T) = \sum_{x_T} \Psi'(x_A, x_L, x_T). \tag{2.21}$$

The final rule is identical to (2.17).

The advantage of the LS algorithm over the simple factoring is that we can reuse the initialized clique potentials for subsequent queries. However, generality has its cost. The join tree size is larger in the LS algorithm and thus computationally more expensive.

The join tree in the LS algorithm might be substantially larger than the join tree in the optimal factoring given enough evidence instantiations in the network. Given that the cases with many evidence nodes are more frequent, the optimal factoring performs much faster and requires much less memory than the LS algorithm in practice.

Despite the differences in the data structure organization in the optimal factoring and LS algorithm (LS algorithm has to store potentials and messages which optimal factoring doesn't), the structure of computations, which we shall consider in more detail in Chapter 3, is identical in both algorithms.

2.1.3 Exact inference complexity

Probabilistic inference has been proved to be *NP*-hard in general and thus is expensive computationally [Cooper, 1990]. The inference time is determined by the total number of states in all cliques in the join tree:

$$\sum_k |C_k| = \sum_k v^{N(C_k)}, \tag{2.22}$$

where we assumed for simplicity that all nodes have the same cardinality v. From the practical point of view, the inference time is determined by the largest clique size. The number of nodes in the largest clique depends on the BN graph, i.e., the number of nodes and density of interconnections in the graph. Finding an optimal factoring that gives a join graph with the smallest cliques is *NP*-complete [Arnborg *et al.*, 1987].

If we know that the largest clique size does not exceed k, there is an exact algorithm that finds the optimal join tree with complexity $O(n^k)$, where n is the number of nodes in the network [Arnborg *et al.*, 1987]. This algorithm is not practical for k greater than 5, however. Shoikhet and Geiger provide a modification of the above algorithm that runs in $O(R \times n^5)$, where R is the number of minimal separators in the graph [Shoikhet and Geiger, 1997]. A minimal separator is defined as a minimal subset of nodes that disconnects two given nodes in a graph. Unfortunately, R can grow very fast and the authors found that already for $k = 10$ and a relatively small BN size of $n = 100$ nodes the computation might take up to 10 hours on a 100 MHz HP/725 workstation (if done efficiently, the probabilistic inference time itself in such a network is about several seconds).

Thus, for practical join tree construction people have to resort to less computationally expensive methods like maximum cardinality search (see, for instance, [Neapolitan, 1990]), simulated annealing [Kjærulff, 1993], or heuristic guided search [Draper, 1995]. These methods do not provide rigid performance guarantees but work quite well for practical problems. In general, the complexity of building the close to optimal join tree increases very fast with the increase of the number of edges in the BN, as does the computation time for probabilistic inference.

Given the join tree structure, which we assume in this thesis is given up front before inference, the problem of computation cost can be attacked from different directions. One of them is to optimize the structure of computations, for example to make approximations during inference.

2.2 Approximate inference

Due to the complexity of exact probabilistic inference, only fairly small models can be computed exactly. The computation time and memory requirements blow up exponentially with the size of a general network. Thus, we need to reduce computation time by making approximations in inference for many practical networks.

Unfortunately, the classification of approximate techniques is much less transparent and clear cut than the classification of the exact inference techniques; there are many more variations of the approximate inference algorithms as well. The algorithms are often on a boundary between different classes.

In general, approximate probabilistic inference can be classified into model reduction and instance generation. In model reduction we reduce a BN structurally to another BN, which is less expensive to compute. The reduction is made in a way to minimize possible errors in the results. In instance generation we approximate the full joint probability sum by a smaller some over some of the entries. The instances are generated in a way to account for the most of the probability mass in the joint probability distribution.

Model reduction can be either static or dynamic (instance generation is almost always dynamic). Static reduction means that the simplification of the model does not depend on the observed evidence or the required precision. Although it is likely to generate large errors for some evidence cases, the reduced model can be statically stored in the computer memory and inference can be done much faster. Dynamic reduction means that the alterations to the original model depend on the observed evidence and/or required precision. Thus, we can dynamically adjust problem complexity to the required precision.

An approximate algorithm can guarantee either rigid bounds on the error, i.e., that the answer to a query is within a certain interval, or probabilistic bounds, i.e., that the answer to a query is within ϵ with probability $1-\delta$, where ϵ and δ are small parameters depending on the model.

2.2.1 Model reduction

We can sometimes reduce a model to one that can be solved efficiently. We distinguish two approaches. The first one is to reduce the size of the model by cutting some edges or nodes. The second one is to reduce an arbitrary BN to one that may be structurally complex but that has other properties that make it effectively solvable.

Structure pruning

The idea in structure pruning is to reduce complexity of the graphical structure of the BN. For example, we can reduce the edges corresponding to weak dependencies in the model [Kjærulff, 1993; Kjærulff, 1994] or remove the edges in certain contexts, i.e., given evidence about values of certain nodes [Boutilier *et al.*, 1996]. Unfortunately, not many dependencies can be removed without substantially affecting the precision of the model, and the error estimation is impossible without a large computation overhead.

Instead of modifying the structure of the BN, we might reduce the cardinality of nodes and therefore the total computational complexity. Wellman and Liu reduce the number of states per variable by merging several states together [Wellman and Liu, 1994]. We will describe this approach in more detail in Chapter 4.

Another technique to build a solvable model over a large set of random variables is a *single fault hypothesis* [Heckerman *et al.*, 1992]. The authors built a medical diagnosis system for the diagnostic of the lymph node diseases based on pathological analysis. Instead of considering the state space of all combinations of diseases, they assume that only one disease can happen at a time. This assumption results in large computation savings and is, in fact, almost true for the lymph node diseases.

The structure pruning can be done dynamically. For example, we know that the influence of nodes in a BN decays exponentially with the distance between nodes. Draper uses this well-known fact in the Localized Partial Evaluation (LPE) algorithm [Draper and Hanks, 1994]. The inference is performed only on an *active* set of nodes chosen in a special way around the query node.

Kjærulff proposed to reduce the state space of the cliques by randomly choosing a subset of states in the cliques for probabilistic inference [Kjærulff, 1995]. Once we choose the subset, we can do probabilistic inference by propagating messages up and down the tree computing the messages over the chosen states only. This technique is close to random state space sampling which we will describe in Section 2.2.2.

Exactly solvable models

Reduction to the large but solvable models is very similar to the above technique, except in this approach the user is after a certain kind of dependencies between variables that facilitate probabilistic inference, not reducing the structure in general. For example, one can effectively bound the joint probability distribution of some networks (*sigmoid* or *causally-independent* BNs) by the joint probability of an exactly solvable analytical model [Jaakkola and Jordan, 1996]. This method is often referred to as a *variational technique*. Unfortunately, the variational technique works only for networks with very weak dependencies between variables.

In [Kozlov and Singh, 1995] the authors simplify the structure of conditional probability matrices by reducing them to two smaller matrices (which has the name QR-decomposition in linear algebra). Unfortunately, we still lose some information about dependencies in this reduction and the reduction itself can be more expensive computationally than the exact probabilistic inference.

2.2.2 Instance generation

The idea of instance generation is to account for the largest terms in the joint probability sums. The rest of the terms can be treated as noise and, in some cases, efficiently bound. The two main sub-categories are deterministic (search) and random (stochastic simulation) instance generation.

Search

Search is a traditional application for artificial intelligence and many people contributed to efficient search techniques. For instance, Poole developed a search technique based on very general principles. His algorithm looks for most likely partial instantiations with large probability mass in the joint probability distribution [Poole, 1993]. While the algorithm can quickly get a good estimate for networks with extreme conditional probabilities, i.e., probabilities close to zero or one, it has to find many instantiations in a general belief network without "extreme" conditional probabilities to approximate an answer with a reasonable precision.

For the above mentioned BN2O network, Henrion developed a special purpose search technique which he called TopN [Henrion, 1991]. It finds an approximate answer and rigid error bounds using the monotonicity properties of conditional probabilities in causally independent BN2O networks.[4] An extension of the TopN algorithm to more complex multi layer causally independent networks exists and is called TopEpsilon [Huang and Henrion, 1996].

Stochastic simulation

On the side of random instance generation we have a large class of algorithms that go under the general name Markov Chain Monte Carlo (MCMC) simulation methods [Rubinstein, 1981]. Stochastic simulation includes such methods as *forward/backward* simulation [Henrion, 1988; Fung and Del Favero, 1994], *likelihood weighting* [Shachter, 1990], and *Gibbs sampling* [Hastings, 1970], which differ by how we generate the values for the variables in a BN.

While the deterministic search algorithms work well when there are only a few entries with a substantial probability mass, MCMC works best in exactly the opposite case when the probability mass only slightly depends on a specific instantiation. If the probabilities are far from zero or one, i.e., are not "extreme", we can prove sub-exponential convergence of the stochastic simulation methods [Dagum and Luby, 1997].

[4] We will consider causal independence and causally independent noisy-OR networks in Chapter 5.

2.3 Continuous variables

So far we considered probabilistic inference with discrete variables. However, many variables in practical systems are continuous. For example, in the “Temperature” BN in Fig. 1.2, the continuous variable temperature x_T depends on the season variable x_S. In general, we need continuous variables to model temperature, position, velocity, altitude, fuel level, oil pressure, concentration of impurities, time, etc. The need to adequately reason with BNs that contain continuous variables in rapidly increasing.

Probabilistic inference in BNs over continuous variables can be done in a very similar fashion. The difference between continuous and discrete variables is that we have to perform integration over continuous state spaces instead of summation over the discrete ones. Like the summation, integration can be multidimensional, which is known to be computationally expensive.

2.3.1 Continuous BNs

Let us consider probabilistic inference with continuous variables only first. In a continuous BN, nodes represent continuous variables and the dependencies between nodes are given by conditional probability *density functions*. The continuous *joint probability density* is then given by a product of all conditional probability density functions.

Only a few continuous probability distributions can be integrated exactly. For example, a BN is solvable if the nodes have a gaussian dependence: a node probability density is a normal distribution where the mean is a linear function of parent values. The joint probability density in the case of gaussian dependencies is known to be a multivariate normal distribution:

$$N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k \det \boldsymbol{\Sigma}}} \exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}{2}\right), \tag{2.23}$$

where $\boldsymbol{x}$ is the vector consisting of N continuous variables $x_1, \ldots, x_N$, $\boldsymbol{\mu}$ is the mean for each of these variables, and $\boldsymbol{\Sigma}$ is the covariance matrix:

$$\boldsymbol{\mu} = \langle \boldsymbol{x} \rangle ; \qquad \boldsymbol{\Sigma} = \left\langle \boldsymbol{x}\boldsymbol{x}^T \right\rangle - \langle \boldsymbol{x} \rangle \langle \boldsymbol{x} \rangle^T . \tag{2.24}$$

The result of integration of a multivariate normal distribution is a multivariate normal distribution, which can be computed in polynomial time.

The multivariate normal distribution has been used in practical systems, for example in Kalman filters. A Kalman filter is a continuous BN defined on an infinite sequence of time slices. A time slice causally depends on the previous one (the conditional probability is a normal distribution). Each time slice represents a distribution over unobservable variables, and we can have one or several noisy observations of the variables in different time slices (with white noise, i.e., the conditional distribution is also gaussian). We can do probabilistic inference since all potentials and messages are multivariate normal functions (2.23).

In general, if a clique potential is a multivariate normal distribution, the marginal of the potential is also a multivariate normal distribution. Thus, the property of being a multivariate normal distribution is closed under the operations of probabilistic inference and we can do inference by message passing as in the discrete belief networks.

2.3.2 Hybrid BNs

If a network has discrete variables also, we call such a network a *hybrid* network. The join tree in a hybrid networks will contain cliques with mixed discrete and continuous variables. While computing messages, we have to sum over discrete variables and to integrate over continuous variables.

An important class of hybrid network is a Conditional Gaussian (CG) network with dependencies between nodes expressed as a gaussian dependence conditioned on discrete variables. The assumption is that all continuous variables have a gaussian probability density: a node probability density is a normal distribution whose mean is a linear combination of the continuous parents for any given state of the discrete parents. However, the coefficients in the linear expansion as well as the variance can depend on the state of the discrete parents [Lauritzen and Wermuth, 1989].

CG BNs are exactly solvable[5]. Unfortunately, the "conditioned gaussian" dependency is quite limiting: for example, we cannot model a discrete child of a continuous variable with a CG dependency, which happen very often in practical models. For example, we cannot model a switch which is turned on by temperature or pressure.

2.3.3 Approximate inference in hybrid BNs

An extension of the CG technique was to decompose an arbitrary functional dependence as a mixture of CG functions; the clique potentials and messages are represented by weighted mixtures of the multivariate normal distributions [Driver and Morrel, 1995; Alag and Agogino, 1996].

The major deficiency of this technique is that the number of terms in the mixtures can increase exponentially with the propagation length and probabilistic inference quickly becomes intractable. The decomposition of an arbitrary function as a sum of weighted Gaussian also presents a challenging computational problem.

2.4 Conclusions

The major goal of this chapter was to give an introduction to the quickly developing field of probabilistic inference methods. In this chapter, we have chosen an optimal factoring approach to probabilistic inference. Based on the optimal factoring approach, we show that probabilistic inference in a discrete only BN can be reduced to the summation of the joint probability distribution defined by the BN. Analogously, probabilistic inference in networks with continuous variables can be reduced to multidimensional integration.

We consider different techniques to approximate probabilistic inference and provide our classification of approximate probabilistic methods into model reduction and

[5] More exactly, one can exactly find the means and variances of all continuous variables and the exact probabilities of the discrete variables. The up propagation in these networks is done by first integrating over all continuous variables, and then by summing over all discrete variables. The down propagation is done by approximating the mixtures of gaussians by one gaussian, which preserves means and variances of the final distributions [Lauritzen and Wermuth, 1989; Lauritzen, 1992].

instance generation. In the model reduction, we try to reduce the original model to a smaller one or to the one that is known to be efficiently solvable. In the instance generation, we look, either deterministically or stochastically, for the largest contributions to the joint probability sums.

We show the relation of this summation/integration technique to the LS type computations, the most popular algorithm in the past. We have shown why the optimal factoring results in large computational speedup of probabilistic inference and why probabilistic inference is still computationally expensive.

Chapter 3

Parallel implementations

"Although some of our representations, for example differential equations, were developed for the human mind to read, the same representation is now being used by parallel processors. If parallel processors are going to solve this and other problems, the problems' representation should be designed for these machines."

Michael J. Flynn, IEEE Computer, Dec 1996, p152

One of the obvious ways to speed up probabilistic inference is to use fast parallel computers in which we divide the inference computations between several processors. In doing so, we need to reformulate the original algorithm and split the computer workload into several almost independent tasks. The two critical issues for a successful parallelization are *load balance*—how evenly the task workload is distributed between processors, and *data locality*—how well the tasks reuse the data and avoid interprocessor communication. Simultaneous optimization of the two requirements is often infeasible and we have to find a compromise between the two issues depending on the particular instance of the problem at hand.

Probabilistic inference is a particularly interesting application from the point of view of this tradeoff since it is very data intensive. In particular, it is different from the traditional scientific applications, like linear algebra algorithms or hierarchical N-body methods, which were parallelized on the shared address space machines [Rothberg, 1993; Singh, 1993]. While the number of operations per byte of data grows as $O(n^{3/2})$ in many linear algebra algorithms and as $O(n \log n)$ in hierarchical N-body methods with large constants, the number of operations in a probabilistic inference algorithm grows at the same rate as the number of bytes in memory, and the ratio is typically very small. Although there are potentially other applications—say sorting and some grid solvers—that might do computations in direct proportion to memory requirements, in most of them it is possible to avoid data intensity.[1] It is different for probabilistic inference since we have to allocate the memory for the largest clique, whose size is exponential in the number of nodes in a general BN. It may be that exact probabilistic inference is unique in that it is not possible to reduce the memory requirements.

As we have described in the previous chapter, during probabilistic inference the clique potentials are multiplied by the incoming messages and then summed together to form an outgoing message. It takes only about 2–8 multiplications and a summation per double precision number represented by 64 bits in the computer memory to proceed to another clique. Thus, we are bound to have a large number of memory accesses per processor instruction. All problems associated with data locality, e.g., the performance of hierarchical memory systems and the above tradeoff between data locality and load balance in a parallel implementation, are sharpened in a probabilistic inference application.

In this chapter, which is an extended account of [Kozlov and Singh, 1996b], we show that trading some forms of concurrency for enhanced data locality improves the speedup on a multiprocessor and that small perturbations of the program data locality often lead to a drastic degradation in the program speedup. We will notice that parallel probabilistic requires an emphasis on the preservation of data locality,

[1]For example, sorting programs try to do sorting "in place" which makes sorting $O(n \log n)$, where n is the number of bytes of memory required by a program. Grid solvers tend to iterate over the same data multiple times and thus perform many operations per byte of memory.

even at the expense of load balance in some cases. Fortunately, a sufficiently good load balance in probabilistic inference is easily achieved with a simple static partitioning scheme for practical networks, say the CPCS network. If there is a practical network in which dynamic scheme gives substantial advantages in terms of load balancing, it is always possible to augment the proposed static partitioning scheme by dynamic task scheduling and to exploit additional concurrency in the application.

3.1 Uniprocessor implementation

From the efficiency point of view, it is essential to start a multiprocessor implementation with a highly efficient uniprocessor code. For this reason, we start with the description of our optimized uniprocessor code. The results in the rest of this chapter are compared to the results obtained with this optimized code.

Our uniprocessor implementation was derived from an Ergo implementation provided to us.[2] In our original experiments for large BNs like CPCS, a processor (Sun SPARC-20 workstation) spent about 50% of its execution time on memory stalls. Thus, memory management proved to be important even for uniprocessor code. Let us understand the structure of computations and how we can improve the memory system performance.

3.1.1 Structure of computations

For example, let us examine the computation for the query $p(D)$ in Eqs. (2.6 – 2.11) illustrated in Fig. 2.1. First, the C_1 clique potential is formed by multiplying the conditional probabilities $p(x_T|x_V)$ and $p(x_V)$, which is a total of four entries. Then, pairs of the C_1 potential array entries, a pair for each fixed value of the variable x_T, are summed together to form a message to the clique C_4 which is a function of x_T

[2] The standard Ergo implementation of probabilistic inference is courtesy of Adam Galper from Stanford Medical School.

(see Fig. 2.1):

$$m(x_T) = \sum_{x_V} p(x_T|x_V)p(x_V). \tag{3.1}$$

Analogous summation over the variable x_X is carried out in the leaf C_2.

The clique C_4 potential is formed by the conditional probability $p(x_A|x_T, x_L)$ multiplied by the messages $m(x_T)$ and $m(x_A)$ received from the cliques C_1 and C_2 correspondingly and summed to form a message $m(x_A, x_L)$ to the next clique C_5:

$$m(x_A, x_L) = \sum_{x_T} p(x_A|x_T, x_L)m(x_T)m(x_A). \tag{3.2}$$

Now, clique C_5 has to get the messages from C_3 and C_4 children.

The clique C_3 potential is formed by the product of conditional probabilities $p(x_B|x_S)$, $p(x_L|x_S)$, and $p(x_S)$. The clique C_3 message to the clique C_5 is:

$$m(x_L, x_B) = \sum_{x_S} p(x_B|x_S)p(x_L|x_S)p(x_S), \tag{3.3}$$

and the clique C_4 message to the clique C_5 was given by (3.2) above.

In its turn, the clique C_5 message to the clique C_6 is the product of the $C_4 \rightarrow C_5$ and $C_3 \rightarrow C_5$ messages (there are no conditional probabilities assigned to the clique C_5) summed over the variable x_L.

Finally, the clique C_6 potential is formed by the conditional probability $p(x_D|x_A, x_B)$ multiplied by the message from the clique C_3. The answer to the query, $p(D)$, is obtained by summing the clique C_6 potential entries corresponding to the values $x_D = true$:

$$p(D) = \sum_{\{x_A, x_B\}} p(x_D = true|x_A, x_B)m(x_A, x_B). \tag{3.4}$$

The structure of computations is schematically represented in Fig. 3.1, where the potential array entries are denoted by squares and the message array entries are denoted by circles.

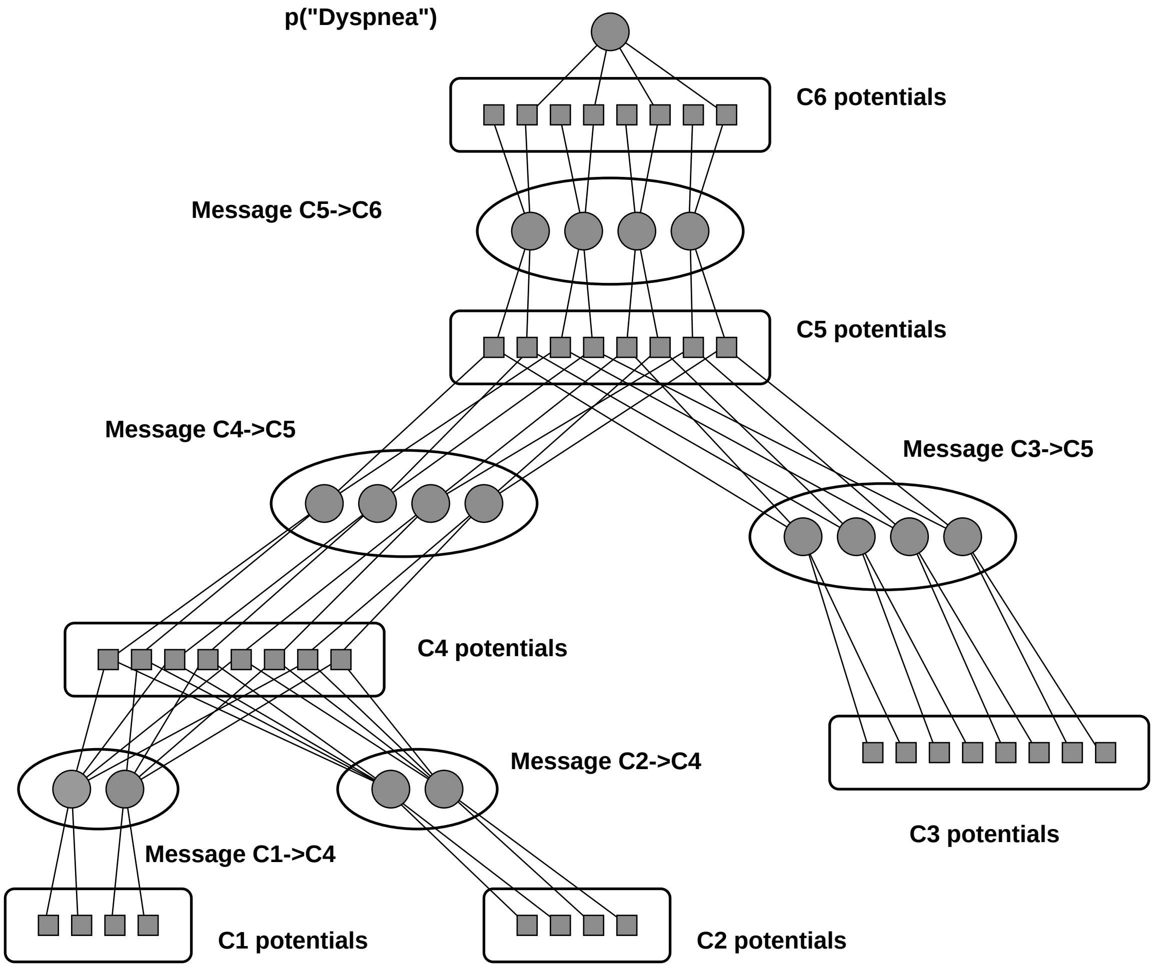


Figure 3.1: Structure of the $p(D)$ query computation in the "Chest Clinique" BN. A clique potential is shown by a rectangle; a message between cliques is shown by an oval. The entries of the potential and message arrays are shown by squares and circles respectively. We assume a row-major layout for the arrays and the following ordering of nodes in the cliques $C_1 = \{x_T, x_V\}$, $C_2 = \{x_A, x_X\}$, $C_3 = \{x_B, x_L, x_S\}$, $C_4 = \{x_A, x_L, x_T\}$, $C_5 = \{x_A, x_B, x_L\}$, $C_6 = \{x_A, x_B, x_D\}$. The number of nodes in the cliques is much larger in practical networks, thus the array sizes are much bigger in practice.

3.1.2 Data locality

Let us examine data locality for the computations shown in Fig. 3.1. In the left branch of the tree, we first sum the C_1 clique potential over the variable x_V. To maximize data locality, we would like to place the potential entries that are summed together close in memory space. We achieve the best data locality by placing the entries corresponding to different values of the variable x_V next to each other, i.e., by making x_V the last index in the array (we assume a row-major layout of the arrays). To maximize the data locality for the $C_2 \to C_4$ message computation, we place the variable x_X last in the array indices for the clique C_2.

The problem is that there is no optimal ordering of nodes for the C_4 potential. If we place the variable x_T first, we achieve the best data locality for the update by the message $C_1 \to C_4$, but not for $C_2 \to C_4$. On the other hand, if we want to achieve the best data locality for the $C_2 \to C_4$ update, we have to place the variable x_A first. However, this compromises data locality for the $C_1 \to C_4$ update. In other words, there is no ordering of C_2 potential array indices that maximizes data locality for both updates. Moreover, computing the message $C_4 \to C_5$ up the tree requires x_T to be the last index of the clique potential array.

Formally speaking, given any two neighboring cliques, the optimal data locality for the update between these two cliques is achieved by placing the variables in the intersection of the node sets, or clique *separators*, first in both clique potential array indices. Thus, we would like to place the nodes that appear in separators first in the array indices, and it is obviously not possible to satisfy for all cliques and all separators. However, we might optimize the data locality following this rule most of the time. Let us consider the join tree structure once again.

One of the major join tree properties, which follows from the optimal factoring approach, is that if a node appears in two cliques it should appear in every clique on the path between these two cliques. Therefore, we can easily compute the number of times a node appears in a separator: it is the number times it appears in cliques minus one. Thus, if we order the nodes in each potential array according to the number of times it appears in cliques—the more frequently a node appears in cliques, the closer it is to the beginning of the potential array index set—we improve the data locality. This ordering of nodes was critical for obtaining the best performance in a uniprocessor as well as multiprocessor implementation.

For example, for the "Chest Clinique" network and the join tree shown in Fig. 3.1, such an ordering is $\langle x_A, x_B, x_L, x_T, x_V, x_X, x_S, x_D \rangle$. Our ordering of nodes resulted in approximately 50% reduction in memory system time compared to a random ordering in the cliques for large networks (with average clique memory requirement bigger than the processor cache size).

3.1.3 Offset evaluation

Another optimization of the original program involved optimization of computations with array indices. To update a potential array entry, the program has to compute the *offset* of the array entry relative to its base address. The offset is computed based on the values of the array multidimensional indices.[3] Given that the number of floating point operations per potential is small, the ratio of the number of operations to compute the offset to the number of operations to do the multiplications and summations of potentials is typically very large; our profiling has shown that up to 90% of clock cycles in the original program is spent on offset evaluation.

To optimize the offset evaluation, we notice that a message array entry updates only those potential array entries of the parent clique that have the same values for the separator variables. For example, the message $C_1 \to C_4$ in Fig. 3.1 depends on the index x_T, which is the separator between cliques C_1 and C_4. Thus, the first message array entry updates the first, third, fifth, and seventh C_4 potential, corresponding to $x_T = \mathit{false}$. Analogously, the second message array entry updates the second, forth, sixth, and eighth C_4 potential, corresponding to $x_T = \mathit{true}$ (see (3.2)).

Thus, the offset evaluation can be divided into two parts: the evaluation of the offset due to the values of variables in the separator, which are also the variables of the message, and the evaluation of the offset due to the rest of the variables in the potential array. The correct final offset is the sum of the two offsets computed

[3] Offset evaluation can be done explicitly, as it was in our program, or by the compiler. We chose to do this computation explicitly since we could optimize it by caching intermediate results.

separately. The computations for the two offsets are repeated for each separate array entry, and the result of these computations can be cached and reused.

For example, in doing the update $C_1 \rightarrow C_4$:

$$p'(x_A, x_L, x_T) = p(x_A, x_L, x_T)m(x_T), \tag{3.5}$$

we first compute the offsets for the two message array entries (common variable x_T), which are 0 and 1 since index x_T is the last, and then compute the offsets due to various combinations of values for x_A and x_L, which are 0, 2, 4, and 6. The correct offsets in the potential array are given by all possible combinations of these two offsets.

The first message, corresponding to offset 0, multiplies the 0, 2, 4, and 6-th potential. The second message, corresponding to offset 1, multiplies the 1, 3, 5, and 7-th potentials. Note, that we do not have to recompute the the full expression for the offset in the potential array in the latter case which would involve x_A and x_L values: we reuse the former offsets and just add one to them. The number of times the offsets are reused is exactly the size of the message array.

Since both offsets are reused, we do not need to reevaluate them for each array entry. The evaluation of the array offset is reduced to the summation of two offsets, both of which we can compute very efficiently. While the computation of offsets contributed up to 90% percent of execution time for the original implementation, it contributed only 20% for our optimized implementation as was measured by `pixie` (i.e., by basic block counting).

3.2 Exploiting parallelism

There are two sources of concurrency in a probabilistic inference algorithm [Kozlov and Singh, 1994]. The first is *topological* parallelism, which corresponds to concurrent processing of different branches of a clique tree, and the second is *in-clique* parallelism, which is due to concurrent processing of the same clique by several processors. Let us give an example: if one processor is working on the clique C_1 and the second on the clique C_2 in Fig. 3.1, we call it topological parallelism. We could also assign the first half of the C_1 clique workload to one processor, and the second half of the C_1 clique workload to another processor, which we would call in-clique parallelism.

We construct and evaluate two parallel implementations of probabilistic inference which exploit the two sources of concurrency at two different levels. One uses both types of parallelism, the topological and in-clique parallelism. Since it is difficult to find the optimal task partitioning in the first implementation that uses topological parallelism—the problem of finding an optimal task partitioning in the join tree topology is an *NP*-hard problem by itself—we select the task partitioning dynamically during the actual propagation. The other implementation uses only in-clique parallelism but significantly improves data locality due to better control over it. We can easily find a sufficiently good task partitioning between processors for in-clique parallelism that preserves good data locality, spatial as well as temporal, during computations.

3.2.1 Task data dependencies

A *data dependency* means that if two computations are performed on two different processors, the processors would have to exchange data. In the absence of data dependencies the work load can be assigned to different processors arbitrarily to satisfy load balance only; with the data dependencies we want to minimize interprocessor data exchange also.

Consider the computation dependencies in Fig. 3.1. A few potential array entries are summed to form a message array entry, which is then used to multiply a few potential array entries of the parent clique. This computation associated with a message array entry is represented by a circle in Fig. 3.2 and is an elementary task that we consider.[4]

A data dependency carried through the clique potentials that computation tasks read and write is represented by an edge. While different entries in the message

[4]Note that a circle in Fig. 3.1 had a completely different meaning, the *message array entry* by itself. The circle if Fig. 3.2 represents a *computation* associated with a message array entry, which involves message as well as potential array entries. Thus, the are no squares in Fig. 3.2, representing a potential array entry in Fig. 3.1.

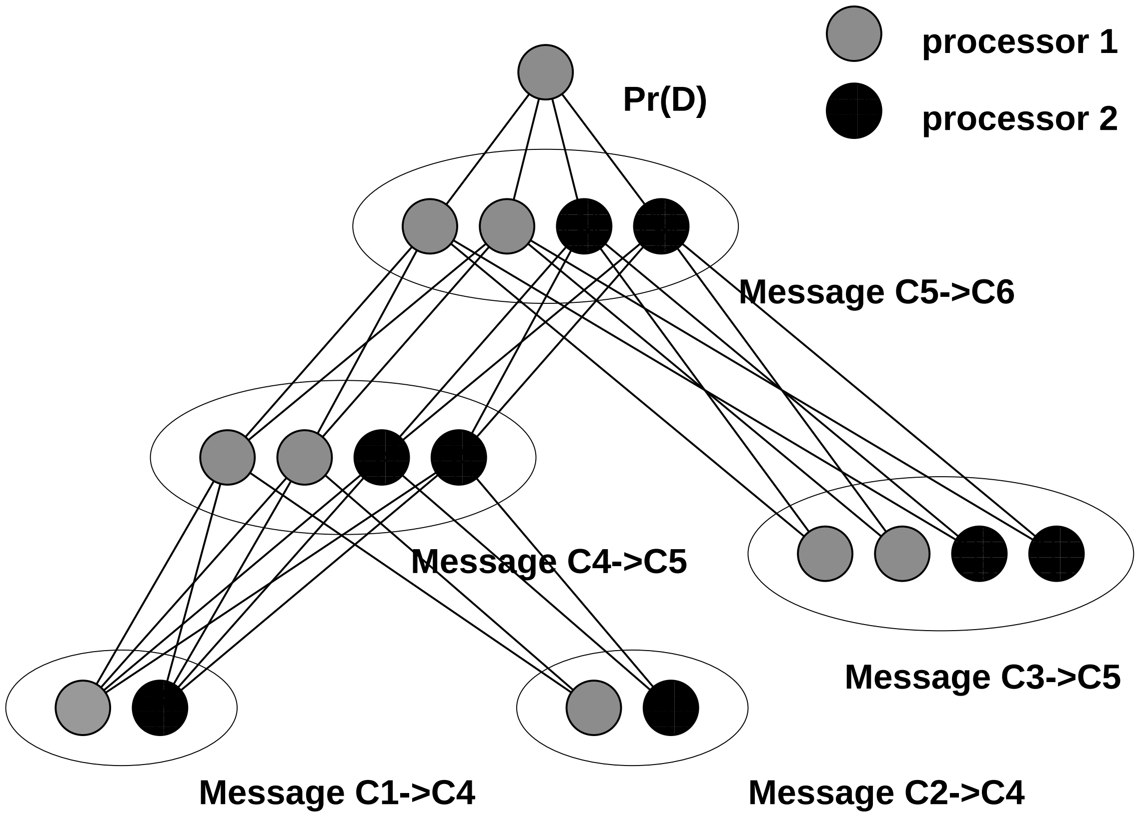


Figure 3.2: The data dependencies between computations associated with message array entries—summation and multiplication of clique potentials—for the evaluation of $p(D)$ shown in Fig. 3.1. A circle represents the computation associated with a single message entry: the summation of the relevant source clique potential array entries to compute an outgoing message entry and the multiplication of the relevant destination clique potential array entries by that message entry (do not confuse with Fig. 3.1). A group of circles in an oval represents a message array. An edge represents a data dependency between the computations. For example, clique C_1 sends a message $C_1 \to C_4$ consisting of two numbers to clique C_4, and all computations associated with the message $C_4 \to C_5$ depend on the computations associated with these numbers through the C_4 clique potentials that they both access. The dependencies between the $C_1 \to C_4$ and $C_2 \to C_4$ messages as well as between the $C_4 \to C_5$ and $C_3 \to C_5$ messages are not shown for simplicity. The computations within a message array are independent. One of the possible assignments of computations to two processors is shown by different shading of the circles.

between two cliques can be computed independently, two processors need to communicate data whenever they perform computations connected by a dependence arc. For example, the computation associated with the first two message entries in the $C_4 \rightarrow C_5$ message is dependent on the computation associated with the first message entry in the $C_2 \rightarrow C_4$ message. This is because the $C_2 \rightarrow C_4$ message entry multiplies C_4 potentials which are then summed to form the $C_4 \rightarrow C_5$ message entries.

The data dependencies for the topological parallelism are simple: the tasks in different branches of the join tree are completely independent until they converge at a common parent. For example, the computations in the cliques C_1, C_2, and C_3 are completely independent. However, the computations for $C_1 \rightarrow C_4$ and $C_2 \rightarrow C_4$ messages become dependent as they come to the common parent C_4.

For the in-clique task partitioning, we can completely avoid interprocessor communication only in special cases. For example, the interprocessor communication is avoided on the path $C_2 \rightarrow C_4 \rightarrow C_5$ for only two processors if we assign the first half of the message to the first processor and the second half to the second processor, as shown by different shadings in Fig. 3.2. More exactly, the first message entry of the message $C_2 \rightarrow C_4$ updates only the first half of the C_4 clique potential, which sum to the first two message entries of the message $C_4 \rightarrow C_5$, which update the first four clique C_5 potential array entries and sum to the first two entries of the message $C_5 \rightarrow C_6$.

In fact, what we observe here is global conditioning [Shachter *et al.*, 1994]. Let us see what happens if we condition the "Chest Clinique" BN on variable x_A. Each of the resulting subnetworks will contain nodes x_V, x_T, x_X, x_L, x_S, x_B, and x_D; to compute the probability $p(D)$ for the subnetwork conditioned on $x_A = \mathit{false}$, we have to perform computations only with potentials and messages for which $x_A = \mathit{false}$, which results in removing the right half of the C_2, C_4, and C_5 cliques arrays corresponding to the $x_A = \mathit{true}$. Correspondingly, to compute the probability $p(D)$ for the subnetwork conditioned on $x_A = \mathit{true}$, we have to perform computations only with potentials and messages for which $x_A = \mathit{true}$, which results in removing the left half of the C_2, C_4, and C_5 cliques corresponding to the $x_A = \mathit{false}$.

In this simple case global conditioning can be viewed as a partial case of our more general task partitioning scheme. Our scheme is equivalent to global conditioning when all cliques on the propagation path contain the conditioning node. In a more general case of propagation in the whole join tree, we get an additional speedup by reusing the $C_1 \rightarrow C_4$ and $C_3 \rightarrow C_5$ messages in our algorithm (which have to be computed in both subnetworks for the global conditioning scheme). The [Shachter *et al.*, 1994] scheme has never been implemented in practice (for parallelization); our implementation is the closest to his original ideas, but also provides significant improvements.

3.2.2 Two task partitioning schemes

The absence of interprocessor communication in the global conditioning suggests a very simple task partitioning. We assign contiguous chunks of the message arrays, ordered according to our heuristic in Section 3.1.2, to different processors. All processors work on each individual clique at a time, and the computations in different cliques are separated by a global synchronization point (a barrier). Since the size of the message arrays in large practical networks is in the thousands, we have enough available parallelism even for this simple static partitioning.

Let us provide an example. To compute the message $C_4 \rightarrow C_5$ in Fig. 3.2, we divide the computation associated with this message between two processors by assigning the first pair of the message array entries to one processor and the second pair to the other processor, or between four processors, by assigning an entry to one individual processor. If the number of processors is larger than four, the computation for each message array entry can be further subdivided among a subset of processors: each processor does an assigned part of the summation. The computation proceeds along linear paths in the join tree, updating the clique potentials on its way. For example, when the computation of the message $C_4 \rightarrow C_5$ is completed, all processors meet at a barrier and continue with the computation of the message $C_3 \rightarrow C_5$ needed to get the correct clique C_5 potential. After another barrier, all processors continue working on the message $C_5 \rightarrow C_6$. The C_5 clique potential is immediately reused, and the temporal data locality is enhanced in this particular type of propagation.

The amount of in-clique parallelism might be limited in small cliques, so in theory certain networks may not perform well with in-clique parallelism only. Our next scheme tries to exploit topological parallelism first and avoids global synchronization at the cost of some data locality. A task in the next scheme is to compute a contiguous chunk of potential/message array entries. When all entries from a potential/message array are computed, we proceed to the next message/potential array up the join tree.

Thus, we can exploit topological parallelism in addition to the in-clique parallelism. The computations with the cliques in different branches are independent and we can assign processor to different branches of the join tree. Since the task load in different branches can be substantially different for unbalanced trees and the join trees are often unbalanced, we had to augment our task scheduling scheme by dynamic load balancing and an initial task assignment heuristic.

The first goal is to assign all tasks in the leaves to individual processors (if this cannot be done evenly then some processors share leaves by dividing the potential array entries between them as in the static version). When a processors finishes with the leave, it proceeds computations up the tree maintaining synchronization with the other processors in the other branches while merging (each clique and each edge has a separate event flag and a lock). If a processor becomes idle, it steals work (a chunk of message entries) from other processors.

We try to preserve data locality in this dynamic scheme in three ways. First, while a processor chooses messages (or potentials) from the front of its assigned set, other processors steal messages from the end of the set, thus reducing false-sharing of cache lines among processors. Second, as far as possible a processor steals work from another processor which works on a clique in the same branch of the join tree, thus ensuring that a group of processors is likely to work on a single branch. Third, clique data are allocated in the local memories of the processors on which they are most likely to be computed (this can be done deterministically for the leaves, but not for the internal nodes of the join tree).

However, the processors eventually need to communicate data when one of them steals work from another processor or when two or more processors update or access the same parent clique during their traversal. For example, if entire children or subtrees of a parent clique are assigned to different processors—the best case for topological parallelism—they don't communicate at all until they reach the common parent, but at that point many of them are likely to communicate when they update the parent's potential. This problem occurs quite often toward the root clique of the join trees and is the main deficiency of the dynamic scheme since in many practical networks workload in concentrated in the root clique.

3.3 Results

We first describe the multiprocessor platforms and the input belief networks we use, and then discuss parallel performance results including the measurements of spatial and temporal data locality.

3.3.1 Multiprocessor Platforms

We ran our multiprocessor experiments on three machines that support an implicit, shared address space communication abstraction as well as coherent caching in hardware. The shared address space abstraction greatly simplifies the parallel programming task, particularly for applications with irregular data access patterns. Parallel programs we developed are written in C using the `parmacs` macro package from Argonne National Laboratories for parallelism constructs.

The Stanford DASH Multiprocessor

The DASH machine [Lenoski *et al.*, 1990] is an experimental multiprocessor built at Stanford University. The machine we used has 32 processors (33 MHz clock speed) organized in eight clusters.[5] A cluster comprises four MIPS R3000 processors and 32MB local memory connected by a shared bus, and clusters are connected together in a mesh network. Every processor has a 64KB first-level cache memory and a 256KB second-level cache, and every cluster has an equal fraction of the physical memory

[5] The prototype actually has 64 processors in sixteen clusters, but is broken up into separate machines in usual operation.

on the machine. That is, memory is logically shared but physically distributed, and four processors share a local main memory unit. The caches are kept coherent across clusters in hardware using a distributed directory-based protocol [Lenoski *et al.*, 1990].

A read miss in the first-level cache that is satisfied in the second-level cache has a stall time of 12 processor clock cycles. A miss in the second-level cache satisfied in the local clusters memory—which we call a *local* miss—results in a stall time of 29 cycles. Finally, if the data are in some remote cluster, the cluster initiates a request on the mesh network and the processor stalls for 100 or more cycles depending on the location of the data, the state of the directory, and contention. We call this miss a *remote* miss.

The SGI Challenge Multiprocessor

We also examine the parallel speedups obtained on commercially available machines with faster processors but centralized main memory. First, we used an SGI Challenge XL machine with 512 MB of shared memory. It has sixteen MIPS R4400 processors (100 MHz clock speed) connected to a shared bus. A snoopy bus protocol is used to keep caches coherent. Compared to DASH, it has a bigger cache line size (128 bytes as opposed to 16 bytes), smaller first-level caches (16 KB as opposed to 64 KB), and larger second-level caches (1 MB as opposed to 256 KB for DASH) per processor. Since all the processors are connected to a single bus, the difference between memory latency due to a second-level cache miss satisfied in the main memory and one satisfied in one of the other processor caches is not large (100 versus 130 processor cycles without contention, respectively). There are no remote misses, data distribution is not an issue, and interprocessor communication is not much more expensive than capacity misses.

The SGI Origin 2000 server

Finally, we compute some of the largest BNs available for our experiments on the Origin 2000 server. The architecture of the commercial Origin 2000 machine is very close to the Stanford DASH except that the clusters have only 2 processors each.

We used an SGI Origin 2000 machine with sixteen MIPS R10000 processors (195 MHz clock speed). It had 16 GB of shared memory and 4 MB of secondary unified instruction/data cache. The size of the first-level level caches was 32 KB. The cost of a local cache miss is 320 ns, while the cost of a remote cache miss depends on how many routers are traversed ($485 + n \times 100$ ns, where n is the number of routers). In general, it has approximately the same as DASH ratio of remote to local miss cost of 1/3, which is quite representative of modern efficient cache-coherent machines.

3.3.2 Networks

network	n	e	v	c	l	$\max\lvert X_i\rvert$	$\max\lvert X_i \cap X_j\rvert$	Memory	Time
RND1	54	2.4	2	30	12	16 M	1 M	164	260
RND2	54	2.2	2	32	16	512 K	256 K	23	33
RND3	18	2.2	4	8	4	1 M	64 K	10	5
RND4	54	2.0	2	30	14	512 K	64 K	8	8
TL8	16	4	4	9	7	256 K	64 K	16	10
MC8	16	4.5	4	9	1	256 K	64 K	16	10

Table 3.1: Parameters of the BNs we used to test the multiprocessor implementation. n: number of nodes, e: average number of edges per node, v: number of values per node, c: number of cliques, l: number of leaves in the clique tree, $\max|X_i|$: maximum size of a potential array in double precision numbers, $\max|X_i \cap X_j|$: maximum size of a message array in double precision numbers, Memory: global memory requirement in MB, Time: uniprocessor (DASH) propagation time in seconds.

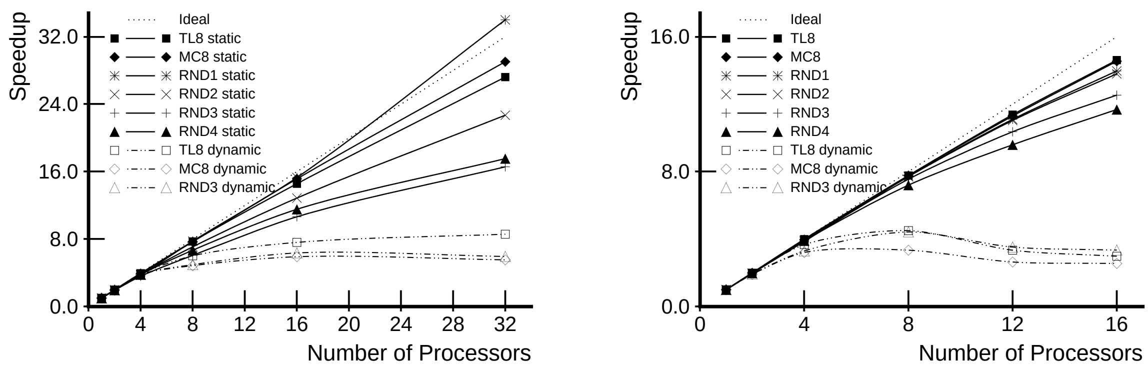
We used six BNs represented in Table 3.1 for our performance measurements. The size of all but the RND1 network was chosen to fit in the memory of a single DASH cluster. Larger networks tend to exhibit better speedups and can even have super-linear speedup, i.e., more than the number of processors, when executed on more than 4 processors on DASH. The first four networks—which we call the RND1 through RND4—were generated randomly. We first built a completely interconnected graph of the given number of nodes and then removed edges randomly until it had the required number of edges. The computation time for these networks with the optimized uniprocessor program varies from 5 to 260 seconds and the global memory requirements range from 8 MB to 164 MB. As we will see later, the random networks exhibit some common as well as some distinguishing data locality properties.

The two other networks have been constructed to resemble networks that are often used in practice. The TL8 network is a two-layer network with eight four-valued nodes in the first layer and eight four-valued nodes in the second layer. Each node in the second layer has all eight nodes in the first layer as its parents. Two-layer networks are often used in diagnosis problems, such as medical diagnosis in which one layer consists of diseases and the other of findings. The MC8 network is a Markov chain network consisting of sixteen four-valued nodes. The sixteen nodes form a chain. However, each of the nodes additionally depend on its eight predecessors. The join trees for the TL8 and MC8 networks consist of eight cliques with nine nodes in each, requiring 262,144 potentials (2MB of RAM per clique when represented by double precision numbers). While the join tree for the MC8 network is a linear chain of eight cliques, the join tree for the TL8 network can be represented as a tree with seven branches, thus exhibiting additional topological parallelism that can be exploited by our dynamic scheme.

3.3.3 Speedups

Fig. 3.3 shows the parallel speedups on the DASH and SGI Challenge machines. The speedup depends on the size and topology of a network, and is generally better for larger networks with larger cliques.[6] The speedups are uniformly better for the static task assignment scheme than for dynamic assignment. And for the static scheme, they are better for the structured networks than for the random ones. To clearly understand the reasons, we measured how the processors spend their time using `UNIX` software profiling tools (`pixie` and `prof`). The resulting decomposition of execution time in Fig. 3.4 indicates that the major difference in parallel performance in all cases is due to the time spent stalled on the memory system. The static scheme was able to reduce communication and exploit data locality at all levels much better than the dynamic scheme, as discussed earlier, and is especially successful when cliques are

[6] The superlinear speedup for the RND1 network will be explained at the end of this section.



(a) DASH

(b) SGI Challenge XL

Figure 3.3: Speedups on the DASH and SGI Challenge XL multiprocessors for both implementations.

large and dependences well-structured.

To explain the increase in memory system time, we measured the relative number of locally versus remotely satisfied second-level cache misses on DASH using a hardware performance monitor. This is a counter installed in the hardware that tracks bus transactions without perturbing the program execution. It sees all second-level cache misses coming from each processor in the cluster, and can tell whether misses are satisfied within the cluster or have to be satisfied remotely (by another cluster). The results for some networks are shown in Fig. 3.5. In the structured TL8 and MC8 networks (random networks have much less symmetry and more random clique sizes) we were particularly successful. The program partitioned the networks with a minimum of cross-processor dependencies and the ratio of local misses (# local / # local + remote) stays close to 100% for these BNs.

For the dynamic partitioning, the ratio quickly becomes much lower, due both to the inherent communication caused by processors picking up dependent tasks unpredictably along the same path in the tree, and to the difficulty of placing data appropriately given this unpredictability of which processors will perform what tasks. Note that since there is one memory per cluster composed of four processors, data

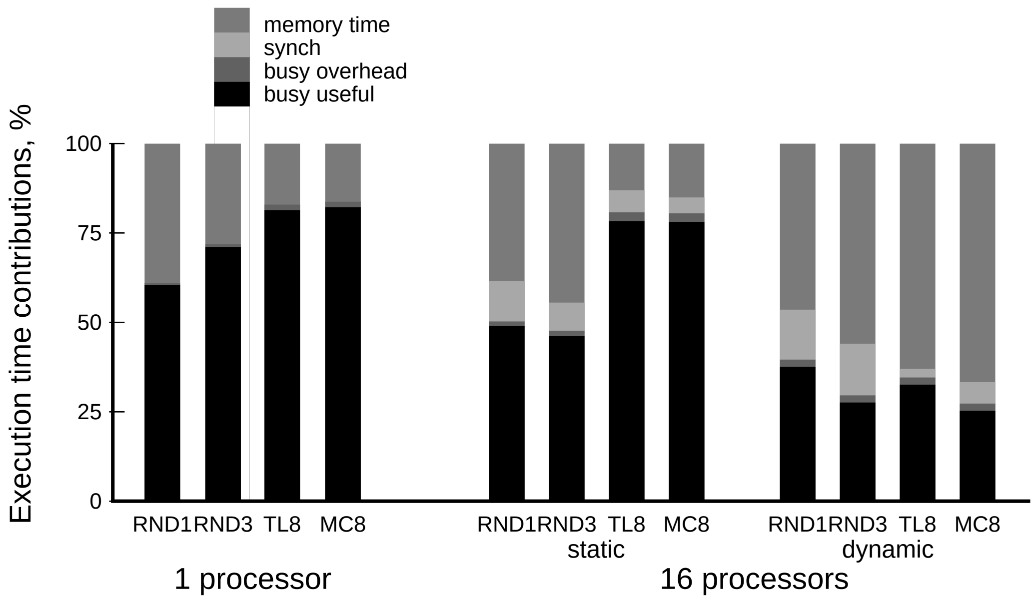


Figure 3.4: Breakdown of the computation time on DASH for the RND1, RND3, TL8, and MC8 networks for both implementations. Busy-useful is the time that the sequential program would spend executing instructions as well. Busy-overhead represents the extra instructions executed in the parallel program. For the static scheme, synchronization time is the time spent waiting at synchronization points. For the dynamic scheme, it includes the time spent on the computations needed to steal tasks, so it does not reflect only load imbalance. The memory time is the time the processors spend stalled on the memory system.

placement and remote misses only become an issue after four processors (if everything fits in one cluster's memory when less than four processors are used). For the static assignment program, the ratio still decreases for random networks like RND3 since the random connectivity of these random networks prevents us from eliminating communication or placing data very well at page granularity; however it decreases less than in the dynamic case. Most real networks have a lot of structure, so we should be able to partition them much better than the purely random networks we generated.

We performed two other experiments which show the importance of data locality and data management in this application. As discussed in Section 3.1.2, our heuristic for node ordering in the cliques improved data locality and decreased memory system time by approximately 50%. In fact, the same ordering helps to avoid interprocessor

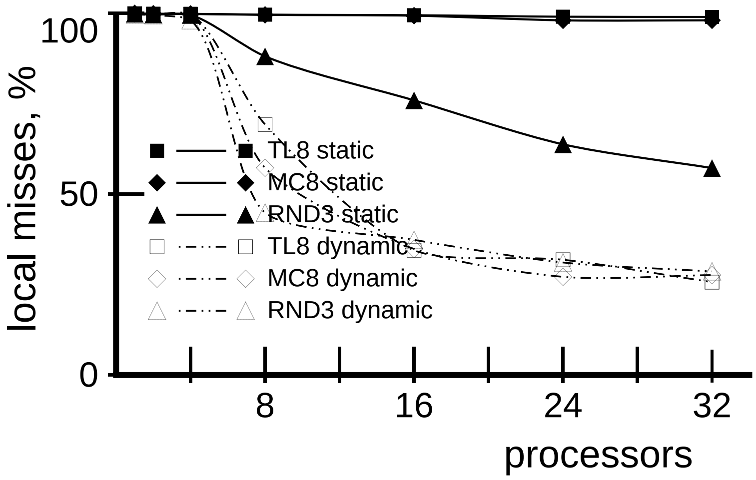
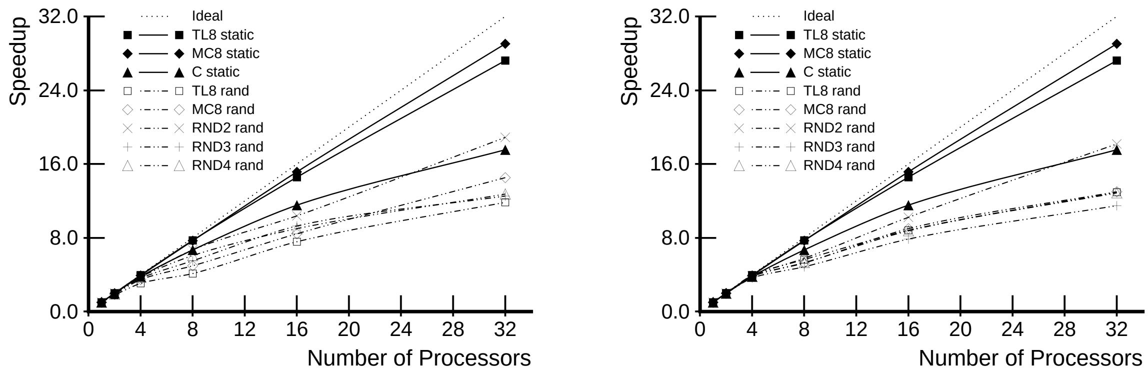


Figure 3.5: Relative number of local misses in the second-level cache for the TL8, MC8, and RND3 for both implementations.

communication for the static multiprocessor implementation (compared to another ordering, say parent ordering). As was shown above, for the $p(D)$ query evaluation in Fig. 3.2, we completely avoid interprocessor communication on the path $C_2 \to C_4 \to C_5$ for two processors. As we discussed, the intrinsic reason for the absence of communication is that x_A is the first index in the C_2, C_4, and C_5 clique arrays and the program performs global conditioning as described in [Shachter *et al.*, 1994]. It is clear that if we perturbed the node ordering, we would have much more interprocessor communication. This is confirmed by Fig. 3.6(a).

The other experiment shows the impact of data distribution across main memories on a distributed-memory machine like DASH. If message entries can indeed be assigned so that there are few cross-processor dependence edges (see Fig. 3.2), and if the entries in a message assigned to a processor are contiguous, pages of data can be placed so that most of a processor's cache misses are satisfied in its local memory. Fig. 3.6(b) compares this allocation strategy with a random page allocation strategy, showing a dramatic performance impact. In fact, the two figures 3.6(a) and 3.6(b) are very similar, although they reflect different types of losses in locality. Proper page



(a) Random node ordering

(b) Random page placement

Figure 3.6: Speedups on DASH compared with those for an equivalent program that uses a parent ordering of nodes within a clique (a) and with an equivalent program that places pages of data randomly among the distributed memories (b). Random node ordering increases communication and compromises data locality (see Fig. 3.1), while random page placement increases the relative cost of capacity misses.

distribution was also implemented for the dynamic version in accordance with the initial assignment of work. As might be expected, the effect is much less pronounced since the dynamic program had worse data locality *a priori*.

The superlinear speedup on DASH for the large RND1 network (see Fig. 3.3(a)) is also due to page placement (not cache effects, as is common). While the uniprocessor execution has to use memory in multiple clusters (the generated data structures do not fit into a single cluster memory), beyond a certain number of clusters the multiprocessor execution uses mostly local memory if pages are placed properly. The reduction in memory stall time from 202 seconds to 160 seconds outweighs the extra 25 seconds spent on synchronization for the 32 processor program execution, resulting in superlinear speedup.

The slightly better speedups on the SGI Challenge than on DASH can be explained by the smaller ratio of latencies corresponding to communication and local capacity misses (130/100 processor cycles for SGI as opposed to 100/29 cycles for DASH) and the larger cache line size (128 bytes for SGI as opposed to 16 bytes for DASH) since, as we shall see, the programs have good spatial locality even for communication (remote) misses due to our ordering heuristic discussed in section 3.1.2. The speedups on DASH are similar to the speedups on the Challenge if the number of processors is less than 4 and the computations are performed in a single DASH cluster.

3.3.4 Data locality

To understand how the temporal and spatial locality in the program scale with the problem size and number of processors, which provides insights into how performance might scale with different systems, we performed software simulations of the multi-processor execution. The simulations with different cache parameters were done with using Tango Light execution-driven reference generator coupled to a memory system simulator [Goldschmidt, 1993]. We modeled a cache-coherent multiprocessor with a physically distributed shared address space and a directory based cache-coherence protocol. The simulator divided cache miss rates according to classification described in [Woo *et al.*, 1995]:

- *Cold* — the first access to a cache line which is not in the processor cache; a cache line must be brought into the cache for the first time.
- *Capacity* — the access to a cache line previously replaced from the cache due to limited cache size; a cache line is brought into the cache for the second time.
- *True sharing* — the access to a cache line which is currently in another processor cache, the next memory request to the given cache line is also done by the same processor.
- *False sharing* — the access to a cache line which is currently in another processor cache, the next memory request to the given cache line is done by another processor causing the cache line to thrash between the processors.

To analyze temporal locality, we measure the working sets of the program by plotting the miss rate versus the size of the per-processor cache used in the simulation, assuming a 16-processor execution and 4-way set-associative caches with a 64-byte

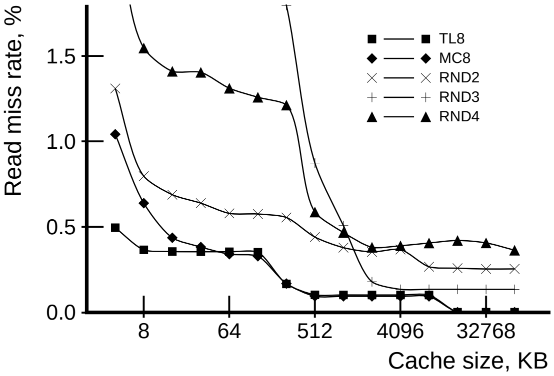


Figure 3.7: Read miss rate as a function of cache size. Simulation for 16 processors and 64 byte cache line size with 4-way set associative caches.

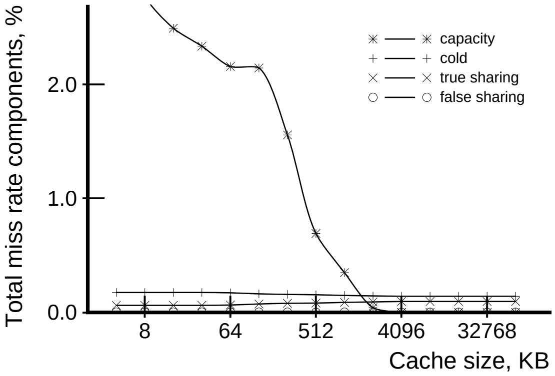


Figure 3.8: Miss rate decomposition into capacity, cold, true sharing and false sharing misses for the RND3 network as a function of cache size. Simulation for 16 processors and 64 byte cache line size with 4-way set associative caches.

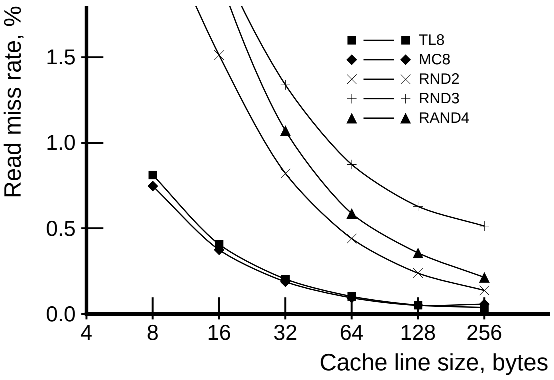


Figure 3.9: Read miss rate as a function of cache line size. Simulation for 16 processors and 512 KB, 4-way set associative cache size per processor.

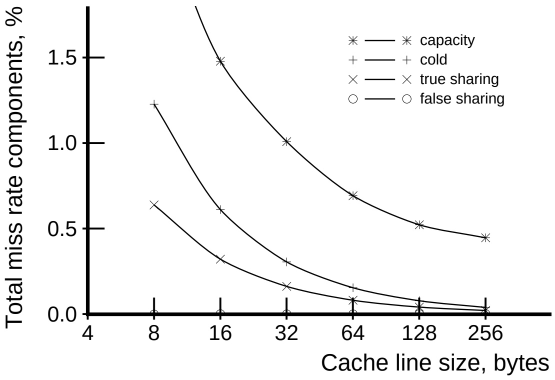


Figure 3.10: Miss rate decomposition into capacity, cold, true sharing and false sharing misses for the RND3 network as a function of cache line size. Simulation for 16 processors and 512 KB, 4-way set associative cache size per processor.

cache line size. The results are shown in Fig. 3.7. For the TL8 and MC8 networks, the working set sizes are approximately the size of one clique memory divided by the number of processors, i.e., 2MB/16 = 128 KB. For randomly generated networks, the dependence is not so distinct since we have a wider distribution over the sizes of the cliques, except for the RND2 network, where we observe only one relatively large clique in which all work turns out to be concentrated (see Table 3.1); the clique size is 8 MB, which produces a sharp rise in the total execution time around 512 KB cache (8 MB / 16 processors). The decrease in the capacity misses, which are the replacements of cache lines due to limited cache capacity, in the decomposition in Fig. 3.8 confirms that the size of the working sets is around 512 KB.

To analyze spatial locality, we measure the dependence of the miss rate on cache line size (the sizes of the processor caches were kept constant at a relatively large size of 512 MB). The results are shown in Fig. 3.9 and 3.10. The dependence characterizes the spatial locality of the program and the amount of data sharing between processors. All components of the miss rate decrease even for random networks as the cache line size increases, including the true sharing or inherent communication miss rate. False sharing of data among processors is insignificant compared to other sources of misses for the randomly generated networks. For the TL8 or MC8 networks, its effect becomes important only beyond a 128 byte cache line. In general, the application has good spatial locality.

3.3.5 Practical applications

We tested our program on one of the largest medical diagnostic networks, the CPCS network (see Fig. 1.3), used for internal medicine diagnosis (the author thanks Dongming Jiang of Princeton University, a student of J.P. Singh, for the help with these experiments).

The version of the CPCS network we used contains 422 binary nodes.[7] The join

[7]The full CPCS network contains 448 nodes (see Fig. 1.3), where some of the nodes have four possible values. The version we had contained 422 binary nodes, and we converted all four-valued nodes to binary. The 422 node CPCS network with four-valued nodes would require about 1 TB of memory.

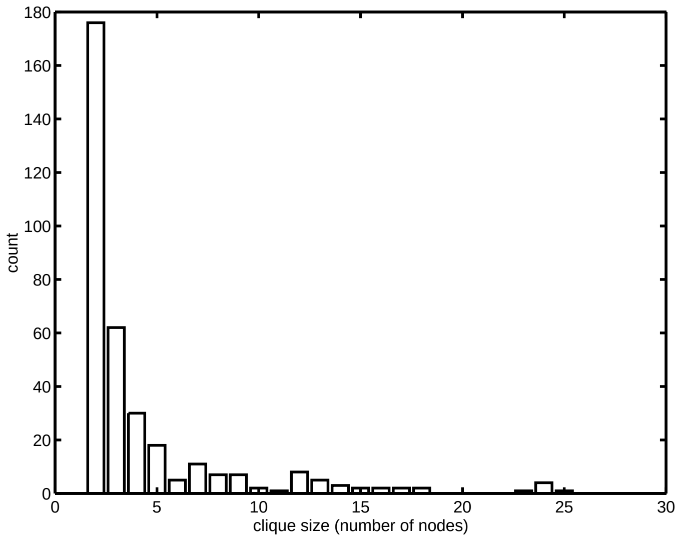


Figure 3.11: Histogram showing the distribution of the clique sizes in the CPCS medical diagnostic network. Although the join tree has many cliques, small cliques contribute to only a small portion of the total workload. For example, the 176 cliques of size 2 contribute to less than 0.0001% of the total workload. Conversely, the six largest cliques—one of 23 nodes, four of 24 nodes, and one of 25 nodes—contribute to 99% of the total workload.

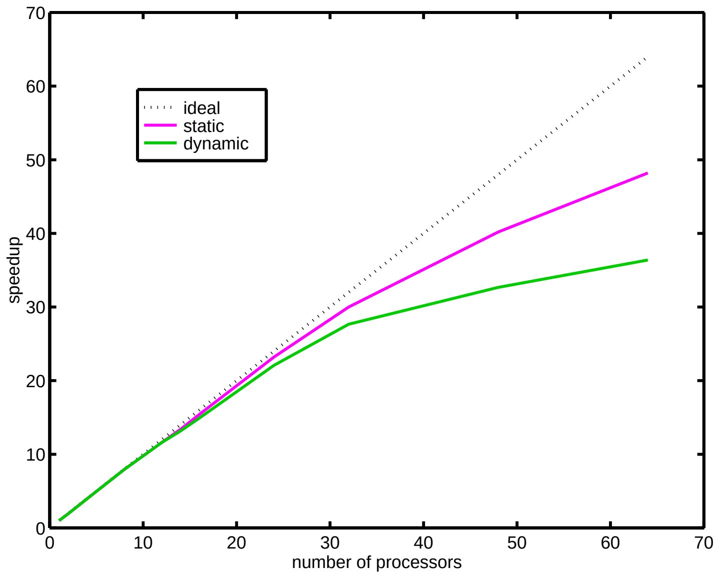


Figure 3.12: Speedups on the SGI Origin 2000 server for static (upper curve) and dynamic (lower curve) implementations.

tree for this network has 350 cliques with total of 110,171,288 potentials. Fig. 3.11 shows the distribution of clique sizes and thus the workload in the cliques (which is exponential in the number of nodes though).

Although we have many more branches and cliques in the join tree as compared to our "artificial" networks and could have expected a high degree of topological parallelism, topological parallelism is still limited in this practical network. For instance, the 176 small cliques of size 4 (2 binary nodes produce a state space of 4) contribute less than $176 \times 4/110,171,288 < 0.0001\%$ to the total workload. Most of the workload is concentrated in a few large cliques, which does not provide enough topological concurrency. The six largest cliques contain $(2^{23} + 4 \times 2^{24} + 2^{25})/110,171,288 \approx 99\%$ of the total workload.

On the other hand, there is enough concurrency on the in-clique level for sufficiently good speedup. Fig. 3.12 shows speedups of our static and dynamic program with the binary CPCS networks. Since the six largest cliques contain 99% of the total workload and the static program exploits this concurrency well, the static program produces good results.

The dynamic program does not perform much worse, however. The join tree has a large fanout: 3 nodes at the second level and 11 at the third. The dynamic program uses the available concurrency on the topological level avoiding expensive global synchronization of the static program. It also successfully exploits available in-clique concurrency where possible.

Let us note that while both programs required about 1 GB of shared memory, the final propagation time in the full CPCS network on 64 processors was less than 10 seconds with our static partitioning program. This low computation time in conjunction with high memory requirements is a direct consequence of the data intensity of probabilistic inference.

3.4 Related work

We have not found any references with an actual implementation of parallel probabilistic inference. Many people discussed possibilities of implementing inference in parallel. Shachter et. al. proposed to condition the BN on several nodes and computing several BNs in parallel [Shachter *et al.*, 1994]. We have shown that our scheme is actually better than the global conditioning scheme. It is exactly equivalent to global conditioning when all cliques contain the conditioning node and is more efficient in other cases.

3.5 Conclusions

We have presented two parallel implementations of exact probabilistic inference, which exploit concurrency at different levels, and have measured and analyzed their performance. The first implementation exploits only in-clique parallelism using static partitioning. The second exploits more of the available concurrency (topological as well as in-clique), using dynamic assignment for load balancing; it avoids global synchronization, but compromises some data locality. We found that the former produces consistently better results than the latter over a wide range of input networks.

Detailed performance analysis shows that the main reason for the better performance of the static scheme is less interprocessor communication and better data locality. The critical forms of locality are data distribution in main memory and spatial locality at cache line level (temporal locality during a single query propagation is limited in optimized implementations). These and reduced communication are obtained by careful ordering and assignment of in-clique computations, and have a large impact on performance. This emphasis on communication and locality makes probabilistic inference a good benchmark for the memory and communication architectures of multiprocessors.

We studied the scalability of data locality with the problem size and the number of processors. We found that important working sets are proportional to clique size divided by number of processors, while good spatial locality persists for a wide range of parameters. Analysis of the algorithm shows that the inherent communication to computation ratio can also be kept low if we increase the clique sizes and the number of processors proportionally.

We ran our two programs on a practical CPCS medical diagnostic network. Although we found that this practical network has much more topological parallelism and that the dynamic program performs almost as well as the static, still about 99% of the total workload is concentrated in six largest cliques. Since our static program utilizes this available in-clique concurrency better, it produces better speedups.

Extensions of the current parallelization schemes are possible for more advanced inference algorithms, for instance the approximate inference algorithms that we consider in the second part of this thesis. These might benefit from extending our initial static assignment with dynamic work stealing for load balance, since the workload becomes somewhat unpredictable even for in-clique parallelism. Overall, the fact that simple parallel implementations can produce good speedups opens up new possibilities for the use of belief networks in decision-support systems.

Chapter 4

State space abstraction

ab-stract, *1542*
Pronunciation: ab-'strakt, 'ab-" (usually in sense 3)
1: *remove, separate;* **2:** *to consider apart from application to or association with a particular instance;* **3:** *to make an abstract of, summarize;* **4:** *to draw away the attention of;* **5:** *steal, purloin.*

Webster dictionary

In practice, it is rarely the case that we need the exact answer for our probabilistic queries—the BNs themselves are often just an approximation of the original underlying problem. The goal of the second part of the thesis is to study another way of reducing the computation time of probabilistic inference through making *approximations* in the inference process. In this chapter, we develop a general approach to making approximations in probabilistic inference based on *abstraction*.

Although approximate probabilistic inference as well as exact probabilistic inference was shown to be *NP*-hard in general and thus computationally expensive [Dagum and Luby, 1993], the computation time to obtain an approximate answer is often much lower than the computation time to obtain the exact answer to a probabilistic query.

The speedup due to making abstractions in inference might be exponential in the size of a general BN by itself (see Chapter 6). Furthermore, approximate methods are often less data intensive, better exploit benefits of the hierarchical memory systems, and thus are more readily accelerated on a multiprocessor.

We start with a formal review of the abstraction paradigm that has been used in physical sciences and decision analysis. We show that similar techniques have been applied to state space abstraction in BNs on the level of state spaces of nodes and parent sets. Then, we describe our abstraction technique in general. The difference between our technique and former ones is threefold. First, we tailor our abstraction to simplify probabilistic inference: we abstract the state space of the whole cliques, not the state spaces of nodes or conditional probabilities. Second, our abstraction is hierarchical: the abstraction at the higher level is expressed in terms of the abstraction at the lower level. Third, we provide a method to dynamically shift between different levels of abstraction during probabilistic inference based on information-theoretic measure of distance, the relative entropy or KL distance often used in the information theory.

4.1 Examples of abstraction

By many accounts, abstraction has been one of the most powerful technique for scientific analysis, particularly in physics and mathematics. For example, when scientists reason about a macroscopic object, they do not need to consider quantum mechanical properties of the macroscopic object parts. Had they had to consider the inner quantum mechanics, they would not have been able to solve even simple mechanical problems, not to mention a problem like satellite scheduling.[1] Fortunately, they can make approximations in reasoning about macroscopic objects by neglecting the lower levels of abstraction, i.e., the quantum mechanical properties of macroscopic objects. In physics, we generally have abstractions on each and every level of the world description.

[1] A cubic centimeter of a typical solid matter contains over 10^{23} electrons; to find an exact behavior of each electron we would have to solve a differential equation in over 10^{23} dimensional space.

4.1.1 Abstraction in decision making

People often use abstraction in everyday life to optimize their decision making. For example, we say that a person "is sick" if he has some evidence of a disease, say fever. At this level of abstraction we do not have to know the origin of the ailment—the sick person is usually confined to bed and taken care of; the lower levels of abstraction might determine patient treatment plan, but are not vital to the initial treatment stage. This abstraction has a very intuitive meaning: when we say that a person is sick we mean that the person has one of several diseases without specifying which one.

Another example is of a car dealer estimating the price of a car. Let us say a car dealer abstracts the car condition by four parameters: condition of the engine, condition of the tires, condition of the interior, condition of the exterior paint, all of which are graded on a scale from 0 to 5. To determine the price of the car, the dealer sums the numerical values for the parameters and prices the car high if it is in excellent condition (the sum is from 16 to 20), medium if the car is in good condition (the sum is from 8 to 15), low if the car is in poor condition (the sum is from 0 to 7).

We have several levels of abstraction in the above example of a car dealer. On the first level, the dealer abstracts the condition in each of the four separate areas by a number from 0 to 5. On the second level, he abstracts the condition of the car in general to price it high, medium, or low. The dealer assumes that all cars in the corresponding range have approximately the same value disregarding the separate contributions to the value.

4.1.2 Abstraction in BN structure

Certainly, abstraction has been used in computer science also. Hoare [Hoare, 1994] writes: "The major achievement of modern science is to demonstrate the links between phenomena at different levels of abstraction and generality, from quarks, particles, atoms and molecules right through to stars, galaxies, and (more conjecturally) the entire universe. On a less grand scale, the computer scientist has to establish such links in every implementation of higher level concepts in terms of lower."

In BNs, abstraction has been most often used on the knowledge acquisition stage; Horvitz & Klein developed a set of general principles for abstraction based on the concepts of *sensitivity* and *utility function* [Horvitz and Klein, 1993]. They construct generalizations about events and actions by considering losses associated with failing to distinguish among detailed distinctions in a decision model. They provide an approach to cluster distinctions into groups of distinctions at progressively higher levels of abstraction and thus show how to transform detailed states of the world into more abstract categories comprised of disjunctions of the states. Finally, they describe rules for decision making with the abstractions.

Abstracting the distinctions between states of the variables has obvious benefits for probabilistic inference: the computation time grows as a polynomial function of the size of the variable domains. If the size of a node domain is v, the state space size of a clique X_i consisting of nodes $x_1, x_2, \ldots, x_n$ grows as:

$$|X_i| = \prod_{i=1,n} |x_i| = v^n, \tag{4.1}$$

as does the computation time. Reduction in the number of states per node reduces the computation time.

Developing the node state space abstraction idea, Wellman and Liu provide an *anytime* heuristic inference algorithm that combines the *elementary states* of variables in *superstates* and performs probabilistic inference in the *abstracted probabilistic network* where one or more variables have superstates [Wellman and Liu, 1994]. For example, if we want to combine two states b_1 and b_2 of a node x_b, the new conditional probability of a node x_a conditioned on the superstate $[b_1, b_2]$ of the node x_b can be computed as:

$$p(x_a|x_b = [b_1, b_2]) = \frac{p(x_a|x_b = b_1)p(x_b = b_1) + p(x_a|x_b = b_2)p(x_b = b_2)}{p(x_b = b_1) + p(x_b = b_2)}, \tag{4.2}$$

according to the definition of conditional probability, and the new conditional probability of the node x_b to be in the a superstate $[b_1, b_2]$ conditioned on some other node

x_c can be computed as:

$$p(x_b = [b_1, b_2]\,|x_c) = p(x_b = b_1|x_c) + p(x_b = b_2|x_c), \tag{4.3}$$

i.e., is the sum of the conditional probabilities for the corresponding states in the superstate. Analogously, for any given superstate $[S_a]$ of the child and any given superstate $[S_b]$ of the parents, the new conditional probability can be found as:

$$p(x_a = [S_a]\,|x_b = [S_b]) = \frac{\sum_{x_a \in [S_a], x_b \in [S_b]} p(x_a|x_b)p(x_b)}{\sum_{x_b \in [S_b]} p(x_b)}. \tag{4.4}$$

Probabilistic inference can be performed much faster with the new conditional probabilities and superstates than with the original complete state space.

There are two problems with computing (4.4). First, the new model no longer satisfies the original independencies and lacks some of the dependencies contained in the original model.[2] Second, the marginals $p(x_b)$ might not be readily available—to get these probabilities exactly we need to perform probabilistic inference in the whole network in the first place. Thus, instead of (4.2), the authors propose to use another approximation:

$$p(x_a|x_b = [b_1, b_2]) = \frac{1}{2}p(x_a|x_b = b_1) + \frac{1}{2}p(x_a|x_b = b_2), \tag{4.5}$$

where we find an arithmetic average of all conditional probabilities instead of the weighted average.

As shown by Wellman and Liu, the results in an abstracted probabilistic network can approximate the exact result in full networks with good precision. The authors start with a very crude abstraction—one or two possible values per node, and proceed by splitting the abstracted states further if time is available. Thus, one can incrementally refine the superstates by introducing distinctions and get a better estimate of the inference result in a BN. The process can be repeated until we either run out

[2]More strictly speaking, the new conditional probability is exact if and only if all original conditional probabilities $p(x_a|x_b)$ are the same for any $x_a \in [S_a]$ and $x_b \in [S_b]$, which would make the separation into the original finer states superfluous in the first place.

of time or solve the original fine grained model. Thus, the algorithm is anytime: the expected accuracy increases as the nodes are iteratively refined.

	p(A)
PV	0.995
$P\bar{V}$	0.985
$\bar{P}V$	0.66
$\bar{P}\bar{V}$	0.01

	p(A)
P *	0.99
$\bar{P}V$	0.66
$\bar{P}\bar{V}$	0.01

Figure 4.1: The context-specific independence (CSI) abstraction for the probability of a car alarm going off. The latter CSI conditional probability can be represented by a decision tree in the upper right corner. The decision tree might be much more compact than the full conditional probability table.

Another proposed way to abstract the state space in a BN is to abstract the states in the conditional probability tables. We can collapse the entries that have almost the same value. This leads to the notion of context-specific independence (CSI) introduced in [Boutilier *et al.*, 1996]. In the CSI conditional probability we have entries that are the same for some different parent values given other parents in certain states, or in certain contexts.

For example, the probability of the car alarm going off, as shown in Fig. 4.1, depends on whether somebody pushes or vandalizes the car with the conditional probabilities given by the table in the lower left corner. We might notice that the first two entries are almost identical, i.e., the probability of a car alarm going off almost does not depend on whether the car is being vandalized if someone is pushing the car. We say that the value of the conditional probability of the car alarm going off is independent of the node x_V in the context that the node x_P in the state *true*.

Thus, the conditional probability $p(x_A|x_P, x_V)$ can be represented by a decision tree shown in the upper right corner (see Fig. 4.1). To find out the conditional probability, the first question we ask is whether the car was pushed, i.e., the value of the variable x_P. If x_P is *true*, we know the answer already (0.99) and do not need to check the value of the other variable x_V; if x_P is *false*, we proceed to ask the value of the variable x_V. The answer depends on the state of x_V and is either 0.66 for $x_V =$ *true* or 0.01 for $x_V =$ *false*. As a result, we end up with smaller data structure to represent conditional probabilities.

Unfortunately, the evidence of computational advantage of the both above schemes is only preliminary. Even though we must have computational advantages due to a smaller state space, the process of abstraction by itself also takes some computation time. Furthermore, finding the error bounds in the above schemes is complicated.

4.2 Clique state space abstraction

Our ultimate goal is to make probabilistic inference more efficient and faster. Thus, instead of reducing the state space of either nodes or conditional probabilities, we propose to reduce the state space of the cliques directly since it is the clique state spaces that determine the computation time of probabilistic inference.

We achieve two major advantages in doing the abstraction on the higher clique level as opposed to the level of separate nodes or parent sets. First, the state space of an average clique is much larger than the state space of a node, and thus we can find more states with similar properties to combine into superstates. Second, there are fewer cliques than nodes or conditional probabilities in a BN—the former are the products of the latter—and we make fewer approximations by abstracting the clique potentials directly.

Just as in the node state abstraction and the CSI abstraction, we combine states with similar characteristics—potential, probability, or conditional probability—into

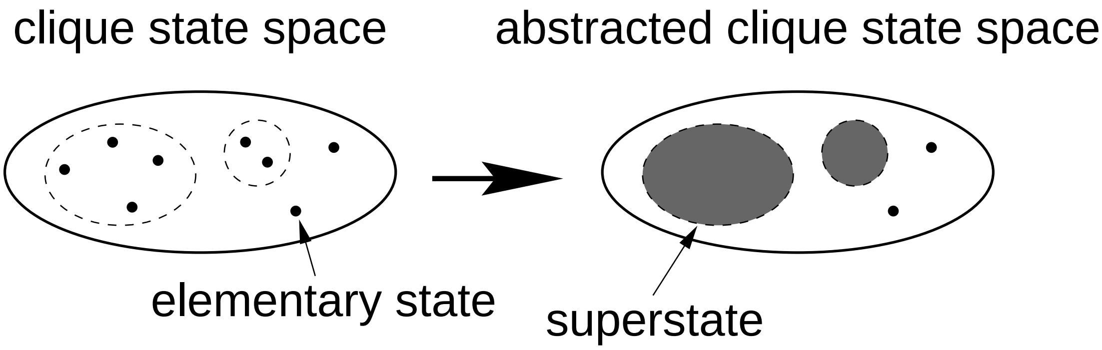


Figure 4.2: The abstraction of the clique state spaces. We combine the states in the clique with similar properties and represent them by superstates. The superstates hold all information about the underlying states and represent the properties of the underlying states on average.

a superstate. The superstate stores the value of the average potential and the information about the states it is comprised of. We will try to modify the probabilistic inference algorithm so that we can perform probabilistic inference operations, the marginalization and multiplication, in terms of the new constructed entities.

We will try to make the abstraction structure convenient for the communication between cliques, i.e., for passing and receiving messages in the join tree. In the node state abstraction this goal was achieved by collapsing the states together across all the cliques. We would like to relax this requirement and make abstraction as flexible as possible in our approach.

4.2.1 Hierarchical abstraction

The simplest way to define a superstate is to assign a subset of states to the superstate denoted as $[condition]$, where the condition might be any condition on the states in a clique. For example, in Fig. 4.1 the condition to combine the states of the parents was $x_P = true$, thus we would denote the superstate corresponding to the two states $\{P\overline{V}, PV\}$ as $[x_P = true]$. In many cases, such as BN2O networks considered in the next chapter, we can find a simple condition to partition the states.

The conditions might be organized into a hierarchy very similar to a decision tree.

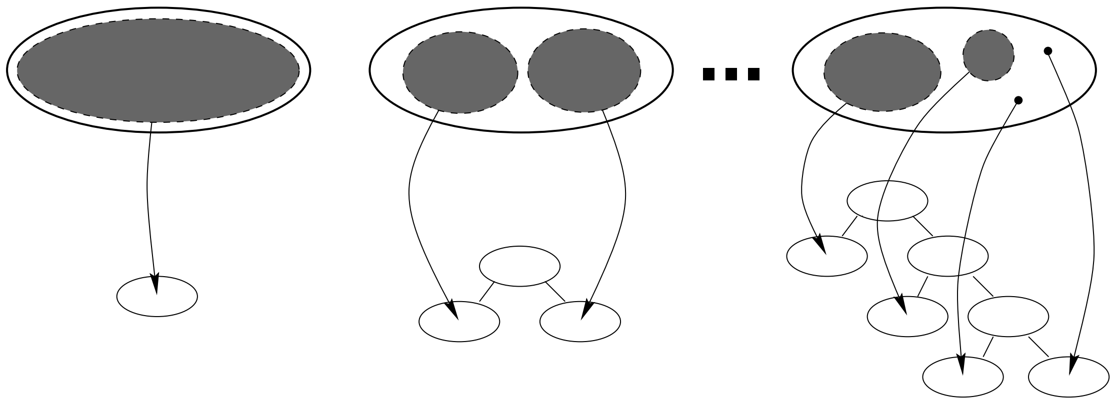


Figure 4.3: A tree representation of hierarchical abstraction.

For example, if we have a state space of two variables, the "top" superstate might contain all possible states for both variables. On the next level, we might split on one of the variables and on the second level on the second of the variable (compare to Fig. 4.1).

A general case of tree hierarchy is presented in Fig. 4.3. We have a few levels of abstraction. The first level abstracts the whole state space in one superstate. The last level of abstraction represents the abstraction in Fig. 4.2. Each parent collapses the superstates of the children into one larger superstate. We find the tree representation of abstraction very convenient for computational purposes.

4.2.2 Dynamic abstraction

While working with a given abstraction, we might find that a given superstate needs to be refined and represented as a collection of smaller superstates. Conversely, if the distinction between different superstates is small, we might combine the superstates and represent them by one superstate on a higher level of abstraction. The hierarchical tree abstraction representation is particularly convenient for such dynamic changes during probabilistic inference since the changes can be localized in one particular branch of the tree.

4.3 Relative entropy, KL and WKL distance

To implement the dynamic abstraction during probabilistic inference, we have to measure the quality of the given hierarchy to see if it needs to be expanded or compressed. The simplest measure of quality is a rigid bound on the difference between the true and abstracted potentials. In a more general setting, we measure the quality by a metric reflecting the quality of the resulting decisions [Horvitz and Klein, 1993].

Measuring the quality of the decisions exactly is a formidable task and is difficult to carry out in practical systems. A simpler metric is the Kullback-Leibler (KL) distance [Cover and Thomas, 1991], an information-theoretic measure of quality. Although both the interval bound and KL distance have been used for estimating the quality of approximation, the KL distance is closer to the real world measure of quality.

In statistics, KL distance arises as an expected logarithm of the likelihood ratio. In encoding theory, it is the difference between the message length ideally encoded according to the true probability distribution p and the message length encoded according to some guessed probability distribution $p_{\mathcal{A}}$. In many other problems, relative entropy translates directly to the expected difference of our utility function due to the changes in probability distributions.

Definition 4.3.1: The *relative entropy* or *Kullback-Leibler distance* $D\left(p \| p_{\mathcal{A}}\right)$ between two probability distributions $p(x)$ and $p_{\mathcal{A}}(x)$ is defined as:

$$D\left(p \| p_{\mathcal{A}}\right) = \sum_x p(x) \log \frac{p(x)}{p_{\mathcal{A}}(x)} = \left\langle \log \frac{p(x)}{p_{\mathcal{A}}(x)} \right\rangle, \tag{4.6}$$

where $\langle x \rangle$ denotes the statistical average of the variable x. ∎

The KL distance is not a distance in the geometrical sense since it does not satisfy the triangle inequality and is not symmetrical; however its properties are similar to the properties of the $\mathcal{L}_2$ distance squared [Cover and Thomas, 1991].

Definition 4.3.2: The *conditional relative entropy* $D\left(p(x_2|x_1) \| p_{\mathcal{A}}(x_2|x_1)\right)$ is the average of the relative entropies between the conditional probability mass functions

$p(x_2|x_1)$ and $p_{\mathcal{A}}(x_2|x_1)$ averaged over the probability mass function $p(x_1)$. More precisely:

$$D\left(p(x_2|x_1)\|p_{\mathcal{A}}(x_2|x_1)\right) = \sum_{x_1} p(x_1) \sum_{x_2} p(x_2|x_1) \log \frac{p(x_2|x_1)}{p_{\mathcal{A}}(x_2|x_1)} = \left\langle \log \frac{p(x_2|x_1)}{p_{\mathcal{A}}(x_2|x_1)} \right\rangle . \quad (4.7)$$

$\blacksquare$

Relative entropy of a probability distribution is conveniently decomposable into a sum of relative entropy and a conditional relative entropy. Suppose that we have a probability distribution $p(x_1, x_2) = p(x_2|x_1)p(x_1)$ defined in a two-dimensional state space $\{x_1, x_2\}$. The relative entropy over the two-dimensional space can be decomposed as follows:

$$D\left(p(x_1, x_2)\|p_{\mathcal{A}}(x_1, x_2)\right) = D\left(p(x_1)\|p_{\mathcal{A}}(x_1)\right) + D\left(p(x_2|x_1)\|p_{\mathcal{A}}(x_2|x_1)\right), \quad (4.8)$$

where $p(x_1, x_2)$ and $p_{\mathcal{A}}(x_1, x_2)$ are some probability distributions, and $p(x_1)$ and $p_{\mathcal{A}}(x_1)$ are the probability distributions over x_1 obtained by marginalization $p(x_1) = \sum_{x_2} p(x_1, x_2)$ and $p_{\mathcal{A}}(x_1) = \sum_{x_2} p_{\mathcal{A}}(x_1, x_2)$.

The relative entropy decomposition has been applied to the task of searching for edges to be removed in the weak-dependencies removal approach discussed earlier in Section 2.2.1. A conditional probability is approximated by another conditional probability with some of the parents removed. The conditional relative entropy:

$$D\left(p'(x_i|Pa(x_i))\|p(x_i|Pa(x_i))\right)$$

between the new and the old probabilities conditioned on the parents $Pa(x_i)$ probability distribution is exactly the additive measure of error we introduce in the joint probability distribution as a result of this substitution. The weak dependency removal has a disadvantage that we have to abstract each of the conditional probabilities and then to combine them to clique potentials which are actually used for probabilistic inference.

Our goal is to do abstraction on the clique level from the start. The clique potentials, however, are not necessarily conditional probabilities and the relative entropy decomposition cannot be applied directly to clique potentials. Thus, we need a new metric to abstract the clique potentials.

4.3.1 Weighted KL distance

Let us see how we can generalize the decomposition (4.8) to a more general case of a joint probability distribution represented as a product of clique potentials:

$$p(x_1, x_2, \ldots, x_n) = \prod_{\text{all potentials}} \Psi(C_k). \tag{4.9}$$

Let us assume that we approximate each clique potential $\Psi(C_k)$ by an abstracted potential $\Psi_{\mathcal{A}}(C_k)$ defined over superstates in the clique C_k. The KL distance between the original joint probability distribution $p(x_1, x_2, \ldots, x_n)$ in (4.9) and an abstracted probability distribution $p_{\mathcal{A}}(x_1, x_2, \ldots, x_n)$ represented as a product of abstracted clique potentials $\Psi_{\mathcal{A}}(C_k)$ is:

$$\begin{aligned} D\left(p(x_1, \ldots, x_n) \| p_{\mathcal{A}}(x_1, \ldots, x_n)\right) &= \sum_{x_1, \ldots, x_n} \prod_k \Psi(C_k) \log \frac{\prod_k \Psi(C_k)}{\prod_k \Psi_{\mathcal{A}}(C_k)} \\ = \sum_{x_1, \ldots, x_n} p(x_1, \ldots, x_n) \log \frac{\prod_k \Psi(C_k)}{\prod_k \Psi_{\mathcal{A}}(C_k)} &= \sum_{x_1, \ldots, x_n} p(x_1, \ldots, x_n) \sum_k \log \frac{\Psi_k(C_k)}{\Psi_{\mathcal{A}}(C_k)} \\ &= \sum_k \sum_{C_k} \frac{\sum_{x_i \notin C_k} p(x_1, \ldots, x_n)}{\Psi(C_k)} \Psi(C_k) \log \frac{\Psi(C_k)}{\Psi_{\mathcal{A}}(C_k)}, \end{aligned} \tag{4.10}$$

where the second sum is over all variables in a clique C_k. The above expression can be rewritten as:

$$\begin{aligned} &D\left(p(x_1, \ldots, x_n) \| p_{\mathcal{A}}(x_1, \ldots, x_n)\right) = \\ &\qquad \sum_k W\left(\Psi(C_k) \| \Psi_{\mathcal{A}}(C_k); \frac{\sum_{x_i \notin C_k} p(x_1, \ldots, x_n)}{\Psi(C_k)}\right), \end{aligned} \tag{4.11}$$

where we introduced a new notation:

$$W\left(p(x)\|p_{\mathcal{A}}(x); w(x)\right) = \sum_{x} w(x)p(x)\log\frac{p(x)}{p_{\mathcal{A}}(x)}. \tag{4.12}$$

Expression (4.12) differs from (4.6) only by the weight function $w(x)$, which was $\sum_{x_i \notin C_k} p(x_1, \ldots, x_n)/\Psi(C_k)$ for the join tree decomposition. Thus, we call it Weighted Kullback-Leibler (WKL) distance between arbitrary positive functions $p(x)$ and $p_{\mathcal{A}}(x)$ with the weight $w(x)$.

4.3.2 WKL distance properties

Let us look at the properties of the WKL distance. First, if the clique potentials $\Psi(C_k)$ are conditional probabilities, the decomposition (4.11) is reduced to the relative entropy decomposition (4.8). Let us see how it can be done.

Consider clique C_k and assume that $\Psi(C_k) = p(x_i|Pa(x_i))$, i.e., each clique is assigned exactly one conditional probability. The marginal $\sum_{x_i \notin C_k} p(x_1, \ldots, x_n)$ used in the weight:

$$w(C_k) = \sum_{x_i \notin C_k} p(x_1, \ldots, x_n)/\Psi(C_k) \tag{4.13}$$

is exactly the prior probability distribution for the variables in the clique. If $x_i \in C_k$, then the ratio $\sum_{x_j \notin C_k} p(x_1, \ldots, x_n)/p(x_i|Pa(x_i))$ is the prior probability distribution $p(Pa(x_i))$ for the parents of the clique. The weight is reduced to the prior probability distribution for the parents of x_i, and thus the weighted relative entropy (4.12) is reduced to conditional relative entropy (4.7) where $x_1 = Pa(x_i)$ and $x_2 = x_i$.

Our final goal is to improve the quality of our decisions or to bound the KL error of our results. The following lemma shows the relation between the WKL and KL distance.

Lemma 4.3.1: *Given the original potential $\Psi(C_k)$, a piecewise constant abstracted potential potential $\Psi_{\mathcal{A}}(C_k)$ which is equal to the average of $\Psi(C_k)$ in each of the abstracted subspaces $[i]$ where it is constant, and a piecewise constant positive function*

$w(C_k)$ which is also constant in all $[i]$, the following bounds hold:

$$\frac{W\left(\Psi(C_k)\|\Psi_{\mathcal{A}}(C_k); w(C_k)\right)}{||\Psi(C_k)||\ \min_j w(C_j)} \geq D\left(p(C_k)\|p_{\mathcal{A}}(C_k)\right); \tag{4.14}$$

$$\frac{W\left(\Psi(C_k)\|\Psi_{\mathcal{A}}(C_k); w(C_k)\right)}{||\Psi(C_k)||\ \max_j w(C_j)} \leq D\left(p(C_k)\|p_{\mathcal{A}}(C_k)\right), \tag{4.15}$$

where $p(C_k)$ and $p_{\mathcal{A}}(C_k)$ are the normalized probability distributions:

$$p(C_k) = \Psi(C_k)/||\Psi(C_k)||; \tag{4.16}$$

$$p_{\mathcal{A}}(C_k) = \Psi_{\mathcal{A}}(C_k)/||\Psi_{\mathcal{A}}(C_k)||, \tag{4.17}$$

and the norm $||\Psi(C_k)|| = ||\Psi_{\mathcal{A}}(C_k)||$ is the total probability mass:

$$||\Psi(C_k)|| = \sum_{x_i \in C_k} \Psi(C_k). \tag{4.18}$$

Proof: To prove a bound on the sum, we have first to show that the contribution to the WKL distance from each group of the abstracted states $[i]$ is positive. The abstracted potential $\Psi_{\mathcal{A}}(C_k)$ is the average of the original potential $\Psi(C_k)$ in each group $[i]$ of the abstracted states:

$$\Psi_{\mathcal{A}}([i]) = \sum_{s \in [i]} \Psi(s)/|[i]|,$$

where $|[i]|$ is the number of primitive states in $[i]$. Due to the convexity properties of the function $x \log x$, we have:

$$\begin{aligned} w \sum_{s\in[i]} \Psi(s) \log \Psi(s) &\geq w \sum_{s\in[i]} \Psi_{\mathcal{A}}(s) \log \Psi_{\mathcal{A}}(s); \\ w \sum_{s\in[i]} \Psi(s) \log \Psi(s) &\geq w \sum_{s\in[i]} \Psi(s) \log \Psi_{\mathcal{A}}(s); \\ w \sum_{s\in[i]} \Psi(s) \log \Psi(s)/\Psi_{\mathcal{A}}(s) &\geq 0, \end{aligned}$$

where we used the fact that $\Psi_{\mathcal{A}}(C_k)$ as well as the weight w is constant within an abstracted state $[i]$.

Let us first replace the function $w(C_k)$ by a constant $\min_j w(C_j)$:

$$W\left(\Psi(C_k)\|\Psi_{\mathcal{A}}(C_k); w(C_k)\right) \geq W\left(\Psi(C_k)\|\Psi_{\mathcal{A}}(C_k); \min_j w(C_j)\right) \tag{4.19}$$

since we can bound the contribution from each separate ω_i. Now:

$$\begin{aligned}\frac{W\left(\Psi(C_k)\|\Psi_{\mathcal{A}}(C_k); \min_j w(C_j)\right)}{\min_j w(C_j)} & \\ &= \frac{\sum_{x_i \in C_k} \min_j w(C_j)\ \Psi(C_k)\log \Psi(C_k)/\Psi_{\mathcal{A}}(C_k)}{\min_j w(C_j)} \\ &= \sum_{x_i \in C_k} \Psi(C_k)\log \Psi(C_k)/\Psi_{\mathcal{A}}(C_k). \end{aligned} \tag{4.20}$$

To convert the last sum to the KL distance, we have to normalize the potentials since the KL distance is defined only on the probability mass functions whose total mass is one. We divide $\Psi(C_k)$ and $\Psi_{\mathcal{A}}(C_k)$ by a constant:

$$||\Psi(C_k)|| = \sum_{x_i \in C_k} \Psi(C_k) = \sum_{x_i \in C_k} \Psi_{\mathcal{A}}(C_k). \tag{4.21}$$

We have:

$$\begin{aligned}\frac{1}{||\Psi(C_k)||}\sum_{x_i \in C_k} \Psi(C_k)\log\frac{\Psi(C_k)}{\Psi_{\mathcal{A}}(C_k)} &= \\ \sum_{x_i \in C_k} \frac{\Psi(C_k)}{||\Psi(C_k)||}\log\frac{\Psi(C_k)/||\Psi(C_k)||}{\Psi_{\mathcal{A}}(C_k)/||\Psi_{\mathcal{A}}(C_k)||} &= \\ D\left(p(C_k)\|p_{\mathcal{A}}(C_k)\right), \end{aligned} \tag{4.22}$$

where we introduced the notations (4.16) and (4.17). Combining the above expressions, we result in the bound (4.14). Analogously, replacing the weight function $w(C_k)$ by the maximum of the weight $\max_j w(C_j)$, we come to the bound (4.15). ∎

Thus, the error corresponding to the KL distance decreases as we increase the weight and/or multiply the potential by a constant factor greater than one.

4.3.3 Weight assignment

To have the correct WKL measure of approximation, we have to know the joint probability marginal (4.13) for each clique, which is typically not readily available during the abstraction process. In general, obtaining the marginal is equivalent to performing probabilistic inference. There are two possible solutions which we consider in the next two chapters.

- Have a guess about the marginals and build our abstraction based on this guess. We use this strategy in Chapter 5 to reduce an arbitrary BN2O network to a polynomially solvable model which produces inference results of good quality. We show that given the assumption of the low prior probability of faults, which is a valid assumption in BN2O diagnostic networks, we can substantially reduce computation time at the expense of a small error.

- Iterate our abstraction and inference mechanism getting better quality guess about the posterior and required abstraction with each iteration. We study the second approach in Chapter 6. We assign an initial guess about the weights on the first iteration and iteratively improve the quality of our marginals and weights on subsequent iterations. We show that the iterative algorithm converges to the correct values and chooses the correct abstraction over the clique state spaces.

4.4 Related work

Trading the precision of the model for computation time in BNs has been proposed, for example, [Horvitz and Klein, 1993]. While the approach the authors suggested is ideal in principle, practical implementation of the proposed approach requires the exact knowledge of modeling choice effects on computation time, a task that might be just as hard as inference itself.

All previous work on reducing the precision of the model was done on the level of nodes (state-space abstraction [Wellman and Liu, 1994]), or conditional probabilities (weak dependencies removal [Kjærulff, 1994] or CSI [Boutilier *et al.*, 1996]). This thesis is the first work that studies how we can lift the approximation, abstraction in this particular case, to the level of cliques, the level where the state space size directly affects probabilistic inference time.

Like in [Kjærulff, 1994] and [Boutilier *et al.*, 1996], we use an information theoretic measure of distance, the KL distance. However, to make it useful for the case of clique potentials, which might or might not be conditional probabilities, we had to generalize the KL distance to the KL distance with weights.

4.5 Conclusions

The contribution of this Chapter is twofold. First, we propose to do abstraction on the clique level as opposed to the level of nodes or conditional probabilities. Second, we generalize the KL distance to WKL distance which can be directly applied to the join tree factoring decomposition.

We generalize the approaches to abstraction in BNs and develop a methodology to abstract states at the clique level, the level which is directly involved in probabilistic inference. We propose to represent such an abstraction by a hierarchical structure on which we will do the probabilistic inference operations, the propagation of messages in the join tree by summation and multiplication, directly. Since there are fewer superstates than elementary states, probabilistic inference can be carried out much faster with the superstates.

We describe a *flat* and *hierarchical* abstraction; flat abstraction can be implemented more efficiently in *static* models while hierarchical abstraction allows to efficiently shift between different levels of abstraction *dynamically*. We analyze the quality of possible state space partitioning in terms of the information-theoretic measure of distance, the relative entropy or KL distance, and describe techniques to choose the weights in the *WKL distance*.

Chapter 5

Static abstraction in BN2O networks

"It is of highest importance in the art of detection to recognize, out of a number of facts, which are incidental and which are vital."

Sherlock Holmes

The first of our trial cases for abstraction is a two layer BN with a noisy-OR interaction between nodes, which is often called a BN2O type network. Many networks of this type are used in practical systems for diagnosis. For instance, the QMR-DT medical diagnostic networks we described in the introduction is a BN2O network with over 4,600+ nodes and is used in practice. The BN2O structure provides a convenient paradigm for knowledge modeling; it makes it easy to construct and use this type of BNs in a diagnostic system.

In this chapter, we develop an approach for static state space partitioning of clique states into superstates and test it on two BN2O networks: one randomly generated and the other generated with the parameters of the practical CPCS network.[1] Although we can define as many superstates as we want, here we explore the use of

[1] The QMR-DT network is proprietary and was not available to us for publication.

only one superstate. Given the analytical properties of the noisy-OR interaction, one superstate per clique results in a polynomial time approximate algorithm for probabilistic inference in the BN2O networks. [Kozlov and Singh, 1996a] provides a slightly different description of this approach based on the concept of *similarity of states*.

5.1 BN2O networks

The structure of a BN2O BN facilitates network construction and probabilistic inference. A simple BN2O network is shown in Fig. 5.1; it consists of two layers of nodes. The nodes in the first layer ("Flu", "Cold", and "Sunburn" in our example) are faults or diseases and usually are unobservable. The nodes in the second layer ("Running nose" and "Fever" in our example) are observable evidence or symptoms. The nodes in the second layer depend on the nodes in the first layer via *noisy-OR* conditional probabilities.

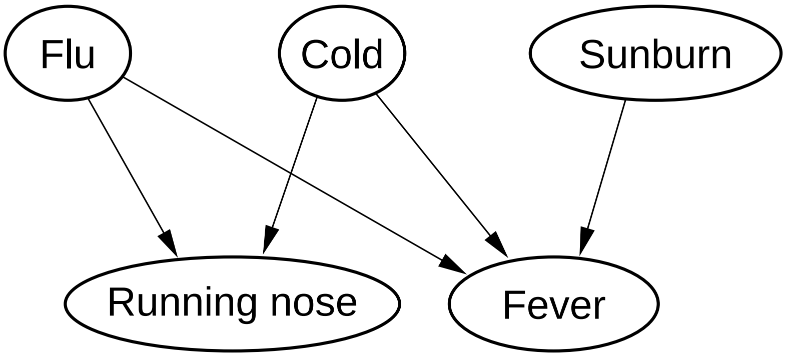


Figure 5.1: Structure of a BN2O network. Flu and cold, but not sunburn, cause running nose; all three diseases might cause high temperature.

A noisy-OR conditional probability is the result of a noisy disjunctive interaction between nodes [Pearl, 1988]; it assumes that the ability of any single disease to cause a symptom is independent of the presence of other diseases. Thus, any probability can be expressed given only a few parameters, which is the number of parents plus one.

5.1.1 Structure of noisy-OR dependencies

Let us start with the simplest example where we assume that the probability of a node when all causes are absent, called the *leak probability*, is zero. For example, we assume that the probability of running nose $p(R|\overline{F}, \overline{C})$ is zero in the absence of flu and cold (see the notations description in Appendix A). Then, two probabilities, the probability $p(R|F, \overline{C})$ of running nose given flu only and the probability $p(R|\overline{F}, C)$ of running nose given cold only, completely define the dependence between the symptom and the diseases. The fourth probability, the probability $p(R|F, C)$ of running nose given both, the flu and cold, can be expressed through the previous two ones.

Numerically, it is simpler to write it in terms of the probabilities of running nose being *false*:

$$p(R|F, C) = 1 - p(\overline{R}|F, C) = 1 - p(\overline{R}|F, \overline{C}) \times p(\overline{R}|\overline{F}, C). \tag{5.1}$$

The conditional probability $p(R|Pa(x_R))$ of "Running nose" to be *true* is a monotonically increasing function: the larger the number of diseases present, the more probable it is for the symptom to appear. Noisy-OR dependence can be expressed more formally based on noisy-OR coefficients.

A noisy-OR dependence between a symptom node x_{s_i} and its n parents x_{d_j} is characterized by $n + 1$ real numbers between 0 and 1: a leak and n coefficients. The leak, which we denote $\mathit{Leak}(s_i)$, is the probability of the symptom being present in the absence of any of the diseases described by the network. A coefficient, which we denote c_{ij}, describes the ability of a disease d_j to cause a symptom s_i in the absence of the other diseases. The rest of conditional probabilities are given by the expression:

$$p(s_i|x_{d_1}, x_{d_2}, \ldots, x_{d_n}) = 1 - [1 - \mathit{Leak}(s_i)] \times \prod_j [1 - c_{ij} x_{d_j}], \tag{5.2}$$

where we assume x_{d_j} to be zero if $x_{d_j} = \mathit{false}$ and one if $x_{d_j} = \mathit{true}$.

Trivially, if c_{ij} is zero, the state of the parent does not affect the probability of the child—there is no edge between x_{d_j} and x_{s_i}. Alternatively, if c_{ij} is one, the *true* state of the parent forces the child to be *true* with probability one—it is a deterministic disjunctive dependence. The noisy-OR interaction model was extended to a noisy-MAX suitable for multivalued nodes [Pradhan *et al.*, 1994].

5.1.2 Inference

Inference in a BN2O BN is polynomial for negative ($x = \mathit{false}$) evidence about nodes and is exponential only in the number of positively ($x = \mathit{true}$) instantiated nodes. Let us demonstrate this.

We consider only evidence in the second layer nodes since evidence in the first layer results in global conditioning of the whole network which can be computed in linear time. If a BN2O network has only negative evidence about symptoms, $x_{s_i} = \mathit{false}$, the disease random variables remain mutually independent (we denote posterior probabilities with a prime):

$$p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) = \prod_{i:x_{s_i}=\mathit{false}} [1 - \mathit{Leak}(s_i)] \times \prod_j [1 - c_{ij} x_{d_j}] p(x_{d_j}), \tag{5.3}$$

where the first product is over the negatively instantiated symptoms. The new posterior disease probability $p'(d_l)$ of a disease d_l can be computed in linear time:

$$p'(d_l) = \frac{p(d_l) \prod_{i:x_{s_i}=\mathit{false}} [1 - c_{il}]}{p(\overline{d_l}) + p(d_l) \prod_{i:x_{s_i}=\mathit{false}} [1 - c_{il}]} \tag{5.4}$$

since we can easy marginalize the product (5.3) representing the joint probability distribution over the disease nodes.

On the other hand, if a BN2O network has only positive evidence about symptoms $x_{s_i} = \mathit{true}$, the posterior probability distribution over the disease nodes is:

$$p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) = \prod_{i:x_{s_i}=\mathit{true}} \left[1 - [1 - \mathit{Leak}(s_i)] \times \prod_j [1 - c_{ij} x_{d_j}] p(x_{d_j})\right] \tag{5.5}$$

and is not decomposable. Thus, the posterior disease probability $p'(d_l)$ can be computed only in time exponential in the number of positively instantiated nodes.

Finally, if the evidence is mixed, we have two types of terms. The terms corresponding to negative evidence can be easily factored out, and the terms corresponding to positive evidence cannot. One of the ways to compute the posterior disease probability in the case of mixed (or positive) evidence is to expand the product as a sum of terms according to the expression:

$$\begin{aligned}(1-a_1)(1-a_2)\cdots(1-a_n) &= 1-(a_1+a_2+\cdots+a_n)\\ &+ (\textit{all possible pairs}) - (\textit{all possible combinations of three a's}) + \cdots ,\end{aligned}$$

where the a's are some coefficients and we sum over all possible singlets, pairs, triplets, etc. The resulting expression for the new posterior disease probability $p'(d_l)$ can be computed in time exponential in the number of positively instantiated nodes and linear in the number of negatively instantiated nodes. If we denote the set of positively instantiated nodes by X_E, the final expression looks like:

$$p'(d_l) = \sum_{x_{s_i} \in X_E} (-1)^{\sum x_{s_i}} \left[\frac{p(d_l) \prod_{i:x_{s_i}=false} [1-c_{il}]}{p(\overline{d_l}) + p(d_l) \prod_{i:x_{s_i}=false} [1-c_{il}]} \right], \tag{5.6}$$

where we sum over all possible combinations of values for $x_{s_i} \in X_E$, which grows as an exponent of $N(X_E)$.

In noisy-OR networks with multiple layers, probabilistic inference can also be sometimes simplified. For example, we can use structural transformations based on the noisy-OR properties as suggested in [Heckerman and Breese, 1994]. These transformations can help to decompose the joint probability distributions into factors for sparsely connected BN2O networks.

5.2 k-fault hypothesis

As shown in equation (5.6), even if we account for the special properties of the noisy-OR, the complexity of probabilistic inference is still exponential in the number of the first-layer nodes (given that we have positive evidence in a general case). Let us see what the join tree for a BN2O network looks like.

For simplicity of presentation, we assume a high interconnectivity of the BN2O network so that the largest clique in the join tree has a size close to the number of nodes in the first layer. The largest clique in a join tree is always larger than the largest parent set; given that the BN2O network is densely interconnected, we might assume that several nodes have most of the first layer nodes as parents. Thus, the largest clique in the resulting tree in a highly interconnected BN2O network includes most of the first layer nodes.

To deal with this and similar situations when the nodes form large cliques, people often use the k-fault hypothesis. The simplest case of the k-fault hypothesis is a single fault hypothesis, which assumes that the faults or diseases are mutually exclusive and thus we assign probability zero to all states s in which $\sum_i x_{d_i} > 1$. A more general k-fault hypothesis states that the probability of all states in which $\sum_i x_{d_i} > k$ is zero.

This assumption greatly simplifies the computations: we need to count only the states with $\sum_i x_{d_i} \leq k$. Thus, the posterior probability of a disease is computed by summing only the "base" states and can be done in polynomial time $O(m \times n^k)$, where m is the number of findings and n is the number diseases. The posterior probability $p'(d_l)$ of a disease d_l is:

$$p'(d_l) = \frac{\displaystyle\sum_{s:\sum_j x_{d_j} \leq k \text{ and } x_{d_l}=true} p(x_{d_1}, x_{d_2}, \ldots, x_{d_n})}{\displaystyle\sum_{s:\sum_j x_{d_j} \leq k} p(x_{d_1}, x_{d_2}, \ldots, x_{d_n})} \tag{5.7}$$

The computational complexity is determined by the number of terms in the above expression which is $O(n^k)$. The single fault hypothesis is a special case of the k-fault hypothesis where $k = 1$.

5.3 Abstraction structure

We choose our abstraction over the state space of the disease nodes X_D, the largest clique in the most general join tree for a densely connected BN2O network. We generalize each conditional probability $p(s_i|Pa(x_{s_i}))$ to $p(s_i|X_D)$. Even though X_D

might contain more nodes than $Pa(x_{s_i})$, we do not increase the computation task in a general case since in a general case all first layer nodes are in one clique.

5.3.1 Computation structure

Let us look first how we can perform probabilistic inference with a superstate which we denote $[\sigma]$. We suppose here that all our conditional probabilities are abstracted, or have the same value for all states $s \in [\sigma]$. First, we compute the contribution of the "base" states $s' \notin [\sigma]$ to the probability of the disease being present:

$$
\begin{aligned}
c'_1(d_l) &= \sum_{s' : s' \notin [\sigma] \text{ and } x_{d_l} = true} p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) \\
&= \sum_{s' : s' \notin [\sigma] \text{ and } x_{d_l} = true} \prod_{j : x_{s_j} = true} p(s_j | X_D) p(x_{d_1}) p(x_{d_2}) \cdots p(x_{d_n}).
\end{aligned}
$$

Analogously, we can compute the contribution of the "base" states to the probability that a disease is absent:

$$
\begin{aligned}
c'_1(\overline{d_l}) &= \sum_{s' : s' \notin [\sigma] \text{ and } x_{d_l} = false} p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) \\
&= \sum_{s' : s' \notin [\sigma] \text{ and } x_{d_l} = false} \prod_{j : x_{s_j} = true} p(s_j | X_D) p(x_{d_1}) p(x_{d_2}) \cdots p(x_{d_n}).
\end{aligned}
$$

The contributions from the superstate could be computed by summing over the underlying states $s \in [\sigma]$ in the superstate:

$$
\begin{aligned}
c_2'(d_l) &= \sum_{s:s\in[\sigma] \text{ and } x_{d_l}=true} p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) \\
&= \sum_{s:s\in[\sigma] \text{ and } x_{d_l}=true} \prod_{j:x_{s_j}=true} p(s_j|X_D)p(x_{d_1})p(x_{d_2})\cdots p(x_{d_n})
\end{aligned}
$$

and

$$
\begin{aligned}
c_2'(\overline{d_l}) &= \sum_{s:s\in[\sigma] \text{ and } x_{d_l}=false} p'(x_{d_1}, x_{d_2}, \ldots, x_{d_n}) \\
&= \sum_{s:s\in[\sigma] \text{ and } x_{d_l}=false} \prod_{j:x_{s_j}=true} p(s_j|X_D)p(x_{d_1})p(x_{d_2})\cdots p(x_{d_n}).
\end{aligned}
$$

The final normalized posterior probability $p'(d_l)$ of the disease d_l is computed as:

$$
p'(d_l) = \frac{c_1'(d_l) + c_2'(d_l)}{(c_1'(d_l) + c_2'(d_l)) + (c_1'(\overline{d_l}) + c_2'(\overline{d_l}))}. \tag{5.8}
$$

So far, our computation is not different from the exact computation and does not have any computational advantage. We still have to sum over an exponential number of states in X_D.

The advantage becomes apparent if we consider that the conditional probability $p(s_i|X_D)$ is the same for all states in the superstate $[\sigma]$. In this case, the conditional probability can be carried out of the summation over the states as a common multiplier. For example:

$$
c_2'(d_l) = \prod_{i:x_{s_i}=true} p(s_i|X_D = [\sigma]) \sum_{s:s\in[\sigma] \text{ and } x_{d_l}=true} p(x_{d_1})p(x_{d_2})\cdots p(x_{d_n}), \tag{5.9}
$$

where the second sum can be usually evaluated in polynomial time. As we might notice, this sum does not depend on the BN instantiation and thus does not depend on evidence. Thus, we define a coefficient $\alpha(d_l)$:

$$\alpha(d_l) = \frac{\sum_{s:s\in[\sigma] \text{ and } x_{d_l}=true} p(x_{d_1})p(x_{d_2})\cdots p(x_{d_n})}{\sum_{s:s\in[\sigma]} p(x_{d_1})p(x_{d_2})\cdots p(x_{d_n})}, \tag{5.10}$$

which is the sum of *prior* probabilities over the group of similar states. The coefficients (5.10) can be computed by either performing the summation over the abstracted states directly, or, if there is only one abstracted state, by the summation over the "base" states:

$$\begin{aligned}
\sum_{s:s\in[\sigma] \text{ and } x_{d_l}=true} p(x_{d_1})\cdots p(x_{d_n}) &= p(d_l) - \sum_{s:s\notin[\sigma] \text{ and } x_{d_l}=true} p(x_{d_1})\cdots p(x_{d_n}); \\
\sum_{s:s\in[\sigma]} p(x_{d_1})\cdots p(x_{d_n}) &= 1 - \sum_{s:s\notin[\sigma]} p(x_{d_1})\cdots p(x_{d_n}).
\end{aligned}$$

Since we want to group as many states as possible in the abstracted state, the number of "base" states might be much smaller, and thus the latter summation can be much more efficient to perform.

Using (5.10), the evaluation of the c_2''s is simplified to:

$$\begin{aligned}
c_2'(d_l) &= \alpha(d_l) \prod_{x_{s_i}\in X_E} p(s_i|X_D = [\sigma])p([\sigma]); \\
c_2'(\overline{d_l}) &= (1-\alpha(d_l)) \prod_{x_{s_i}\in X_E} p(s_i|X_D = [\sigma])p([\sigma]).
\end{aligned}$$

Thus, the model is specified by a few coefficients which determine the contribution of the superstate to each of the disease probability.

5.3.2 Partitioning

Since we know now how to perform probabilistic inference with similar state, let us try to find a simple state space partitioning that provides an abstraction structure that minimizes the error of a general diagnostic query. We can use the WKL decomposition (4.11) for this purpose. The WKL distance between the true $p(s_i|X_D)$ and the abstracted $p_{\mathcal{A}}(s_i|X_D)$ clique potential is:

$$W\left(p(s_i|X_D)\|p_{\mathcal{A}}(s_i|X_D); p'(X_D)/p(s_i|X_D)\right) = \sum_{x_{d_j} \in X_D} \frac{p'(X_D)}{p(s_i|X_D)} p(s_i|X_D) \log \frac{p(s_i|X_D)}{p_{\mathcal{A}}(s_i|X_D)}, \quad (5.11)$$

where X_D is the joint state space of all disease nodes and $p'(X_D)$ is the posterior probability distribution.

In making the partitioning, we have to consider two subgoals:

- To combine states with almost the same potentials, i.e., conditional probabilities, so that we do not alter the original dependencies much.
- To combine states that have a low posterior probability since the sum in (5.11) contains the posterior as a factor.

One obvious way to satisfy both of the above subgoals is to use the monotonic properties of the noisy-OR interaction. The conditional probability of a symptom can only increase with the number of disease present. Since the probability cannot be larger than one, it is likely to be one for many diseases present. Also, like the k-fault hypothesis assumption, we might assume that the posterior for such states is likely to be small (the diseases are unlikely to be present together).

Thus, we combine all states in which the number of disease is larger than k into one superstate $[\sum_i x_{d_i} > k]$. The k-fault hypothesis then becomes a special case where we assign the prior probability of the similar state $p([\sigma])$ value zero. We consider another assignment strategy for the BN2O networks with "extreme" probabilities.

The k-fault hypothesis and our abstraction based on the k-fault hypothesis does not take into account the structure of the problem. If some of the edges between the first and the second layer are missing, i.e., the corresponding noisy-OR coefficient is zero, we can explicitly search for the X_D states that satisfy the above two conditions.

assign all states to the superstate
for each state s **do**
    for each symptom node **do**
        if $p(x_{s_i} = true|X_D = s) < \lambda$ **then**
            retrieve state s from the superstate
        end if
    end for
end for

Figure 5.2: Algorithm for the CPCS superstate selection.

An alternative algorithm for choosing the states in a superstate, which we use in the following section for selecting the abstracted states in a CPCS-like network, is shown in Fig. 5.2. First, we assign all of the X_D states to the superstate. Then, we retrieve the states for which at least one of the conditional probabilities $p(x_{s_i} = true|X_D = s)$ is less than some parameter λ. Thus, we ensure that the states in the superstate have almost the same conditional probability and thus can be abstracted with a better precision (it did not matter much for the randomly generated network since they had no structure in the values of the noisy-OR coefficients).

5.4 Results

We generated two random 20×20 BN2O networks (20 nodes in the first layer and 20 nodes in the second layer). The parameter 20 was picked to be able to traverse all possible instantiations by an exhaustive search over all possible diagnostic queries. The complete traversal time for larger BN2O networks increases exponentially with the number of nodes.

In the first generated network, the noisy-OR coefficients, leaks, and prior probabilities of the nodes in the first layer were generated from a $beta(2,4)$ distribution (with the expected value $\langle c_{ij} \rangle = 1/3$ and variance $\langle c_{ij}^2 \rangle - \langle c_{ij} \rangle^2 = 2/63$), accounting for a skew in the prior symptom probabilities towards the absence.

Although the statistical parameters, the mean and variance, of the beta coefficients are close to what we observed in the CPCS network, the actual values of the

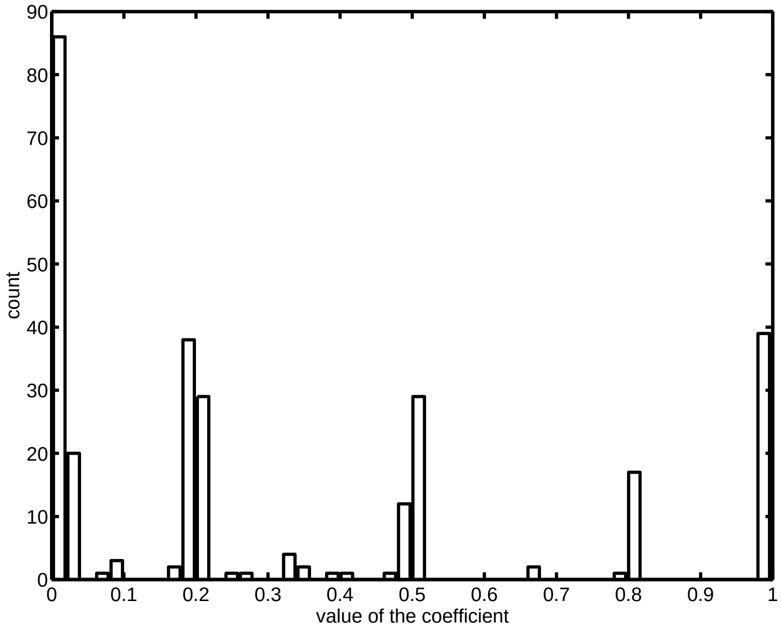


Figure 5.3: Histogram showing the distribution of the noisy-OR coefficients in the CPCS BN. Many coefficients are concentrated around numbers 0 (almost independent nodes), 0.2, 0.5, 0.8, or 1 (deterministic dependence).

noisy-OR coefficients in the CPCS BN are concentrated around some fixed numbers like 0, 0.2, 0.5, and 1 (see Fig. 5.3). To make our experiments more realistic, we also constructed a BN2O network based on the CPCS network. We assigned the parameters of randomly chosen CPCS noisy-OR interactions to the nodes in our 20×20 BN. We used algorithm shown in Fig. 5.2 for the latter network to select the abstracted states. The final number of states in the superstate, and thus the computation time to perform inference, is given by Table 5.1. We call the latter network a CPCS-like network.

We compare the error bounds (absolute and relative) of our model to the k-fault hypothesis model, the best approximate model which has the same computational complexity. For each of the networks we went through all possible positive instantiations of the nodes in the second layer (queries with negative instantiations can be solved using the noisy-OR properties in linear time). We computed the exact posterior probability of each of the diseases and the approximate answer for our model

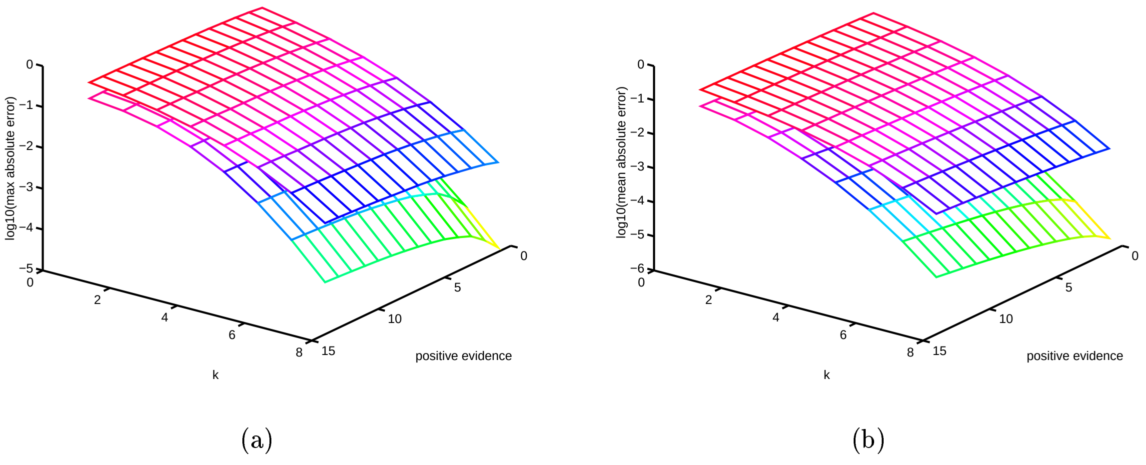


Figure 5.4: Maximum and average absolute errors $\Delta p = |p'(d_l) - p'_{\mathcal{A}}(d_l)|$ of an answer to a query about a disease probability for the abstraction (lower surface) and k-fault hypothesis (upper surface). Maximum as well as average error in the abstraction model is an order of magnitude lower (notice the logarithmic scale along the vertical axis).

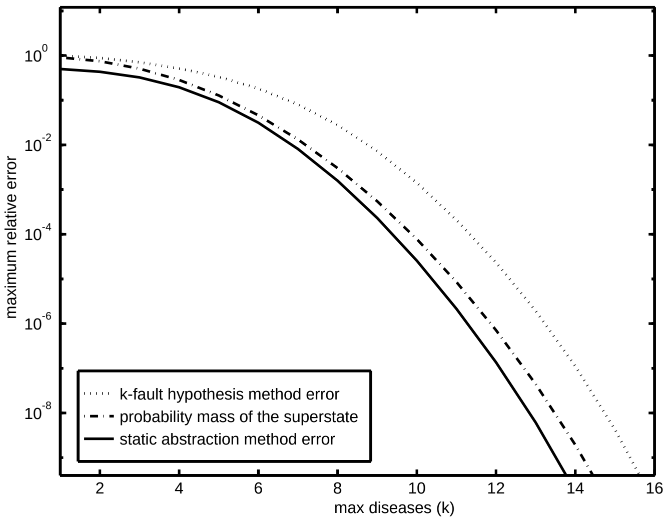


Figure 5.5: The probability of the superstate $p(X_D = [\sigma])$ and the maximum relative error $\Delta p/p = |p'(d_l) - p'_{\mathcal{A}}(d_l)|/p'(d_l)$ of a query result over all possible queries as a function of the parameter k in the k-fault hypothesis method and static abstraction method. All three curves have the same asymptotic behavior. The error in the abstraction method is smaller since it partially accounts for the probability mass that is completely ignored in the k-fault hypothesis method.

and for the k-fault hypothesis.

The maximum and average absolute error for the final disease probabilities in our random networks is about an order of magnitude smaller for our model as compared to the k-fault hypothesis (see Fig. 5.4). The error increases as the amount of positive evidence k increases. As the probability of the abstracted state becomes larger, the abstraction errors become more pronounced. We also notice that the error for the instantiations with the large number of findings present—the region where probabilistic inference is computationally very expensive—is almost independent of the number of instantiated nodes. Thus, we can expect that the approximation scales well to larger problems.

The maximum relative error for the final disease probabilities $p(d_l)$ is also an order of magnitude smaller for our model (see Fig. 5.5). The relative error measure of accuracy might be more appropriate for practical problems. For example, the probability of a life-threatening disease being 10^{-3} is substantially better than the probability of it being 10^{-2}, and the relative error of 10 shows this more clearly than the absolute error of 0.099. The error in the state space abstraction method is about an order of magnitude lower than in the k-fault hypothesis method for high values of k. Our method gives superior precision in more refined models as it partially accounts for the states completely ignored in the k-fault hypothesis method. The maximum relative error is less than 0.01 for $k > 6$ over all possible instantiations of the nodes in the second layer.

The maximum absolute and relative errors for the abstracted CPCS-like network are shown in Table 5.1 for different parameters λ. As we increase λ, the computation time and precision increase. For a relative error of 5% we need to account exactly for only 10% of the total number of states, thus reducing the computation time of the diagnosis by a factor of ten.

5.5 Related work

BN2O networks are practically important; thus, a number of approximate algorithms has been developed for this type of networks. The TopN algorithm [Henrion, 1991] is a

λ	$\lvert X_\sigma\rvert/\lvert X_D\rvert$	Δp	$\Delta p/p$
0.3	95.5%	4.4×10^{-2}	1.8×10^{0}
0.4	89.4%	6.2×10^{-3}	5.4×10^{-2}
0.5	82.6%	5.8×10^{-4}	1.0×10^{-2}
0.6	72.4%	1.3×10^{-4}	1.7×10^{-3}

Table 5.1: Relative reduction in the largest clique state space ($|X_\sigma|/|X_D|$), maximum absolute (Δp) and relative ($\Delta p/p$) errors in the abstracted CPCS-like BN2O network for different threshold selection parameters λ.

search algorithm tailored to the BN2O networks. It looks for the largest contributions to the joint probability sum and gives an upper bound on the error of the final diagnosis. The algorithm got its name since its goal is to determine the N most likely diseases. Since the TopN algorithm does the search during runtime, it is more computationally expensive during runtime than inference in an abstracted network. We believe that a combination of TopN and abstraction is possible where we account exactly for the states found by the TopN algorithm and approximate the contribution from the rest with our technique.

Very often BN2O networks are simulated to obtain the posterior disease probabilities. Stochastic simulation methods have been specifically extended to sample the joint probability distribution of BN2O networks [Henrion, 1988]. In these networks, instance generation can be done very efficiently. The error estimation, on the other hand, is difficult, particularly as BN2O networks often contain probabilities very close to zero. Again, we believe that a combination of the abstraction and stochastic simulation is possible.

5.6 Conclusions

In this chapter, we use general properties of the BN2O model and come to an effective approximate inference algorithm inference. We choose a fixed static state space abstraction that produces small error for the most practical queries. Our abstraction guarantees a sub-exponential computation time.

The model we consider is close to the k-fault hypothesis model used previously. It differs from the latter by how we account for the states with a large number of faults. The k-fault hypothesis discarded them; we try to partially account for them.

We show that by a small additional computation we can significantly improve the precision of the k-fault hypothesis. On the practical side, we show that a reduced model that has only 5% maximum relative error of the diagnosis requires only 10% of the computation time. These results can probably be extended to other BNs with similar structure.

Our work leaves a lot of space for further experimentation. Here, we developed a formalism for one abstracted state (superstate) only. Extension to multiple abstracted states is certainly possible. Our technique can also benefit from a different superstate selection model.

Chapter 6

Dynamic abstraction in hybrid networks

In Chapter 5, we statically constructed an abstracted state in a BN2O network. We assigned a constant value of the clique potential to all states in this abstracted state. Since the potential value was constant, we could efficiently compute the contribution of the abstracted state to a disease probability. The computation was reduced to multiplication of the combined probability mass of the abstracted state by a constant coefficient α.

In the exact model, the contribution from the individual states in the group of abstracted states depends on the evidence. As the probability of the abstracted states becomes larger, the discrepancy between the exact and approximate models becomes more pronounced.

Let us express the above statement formally through the relative entropy decomposition (4.8). We assume that the KL distance between the original and abstracted probability distributions is:

$$\Delta = D\left(p(x,e) \| p_{\mathcal{A}}(x,e)\right) = D\left(p(e) \| p_{\mathcal{A}}(e)\right) + D\left(p(x|e) \| p_{\mathcal{A}}(x|e)\right). \qquad (6.1)$$

The second term, the conditional relative entropy, is represented as a sum (see (4.7)):

$$D\left(p(x|e)\|p_{\mathcal{A}}(x|e)\right) = \sum_{e} p(e) \sum_{x} p(x|e) \log \frac{p(x|e)}{p_{\mathcal{A}}(x|e)}. \tag{6.2}$$

It is exactly the last sum in the above expression, the KL distance between the original joint probability distribution and the abstracted joint probability distribution conditioned on the evidence e, that we would like to minimize. If the probability of evidence is close to one, the last term can be effectively bound by Δ divided by the probability of evidence.

If the probability of evidence $p(e)$ is small, i.e., the query is unlikely, the bound on the error diverges as $\Delta/p(e)$. In practical cases, the answer to a query often has a large error for an unlikely evidence with small $p(e)$.

In this chapter we study another type of abstraction: dynamic abstraction that can adjust itself to evidence and required precision. An unlikely evidence can increase the relative contribution of the subsets of the state space which were abstracted very poorly in the original model. We need to refocus our attention on the abstracted states as they become more and more probable. At some point, as the abstracted state becomes more probable, we want to break it into two or more substates.

We decide whether to break a state using the WKL distance and the weights mechanism developed in Chapter 4. We condition all clique potentials on evidence and abstract the conditioned potentials trying to minimize the WKL distance in each clique. Given that the WKL distance in each clique is less than δ, the error of the final answer is guaranteed to be less than $N \times \delta$ relative to the exact answer, where N is the number of cliques in the join tree (see (4.11)), and does not depend on $p(e)$.

We have chosen to apply this technique to hybrid BNs, BNs that contain continuous as well as discrete variables. Probabilistic inference in hybrid networks is known to be hard; only a few continuous models can be solved exactly without abstraction, e.g., CG networks in which we assume a specific gaussian dependence between nodes (see Chapter 1). If the dependence is far from a gaussian or if we have discrete children of a continuous variable, we have to discretize each and every variable in the network and to perform direct summation (which can also be viewed as direct multidimensional integration).[1]

This approach leads us to one of the most common inference algorithms in hybrid networks with arbitrary dependencies between nodes: to discretize every continuous variable in the network, and corresponds to static state space abstraction on the level of node state spaces. In this chapter, we extend this idea in three ways. First, we perform the discretization on the fly during probabilistic inference. Second, we discretize the whole clique domains at once, not each variable separately. Third, we propose to represent the discretization as a hierarchical structure which greatly simplifies computations with the abstracted potentials.

6.1 Abstraction in hybrid networks

In a discrete network, a superstate represents a subset of discrete states. Likewise, in a network with continuous variables, a superstate represents a continuous interval $a \leq x \leq b$ denoted $[a, b]$ of the continuous variable domain. Without loss of generality, we assume that every continuous variable has domain from zero to one and a clique multidimensional domain is a rectangular n-dimensional hypercube $\Omega = [0, 1]^n$.

6.1.1 BSP tree

We represent a hierarchical abstraction for continuous clique potential $f(\Omega)$ using a tree data structure which we called a Binary Split Partition (BSP) tree. The idea of a BSP tree was borrowed from recursive hierarchical space decomposition used in graphics (see [Samet and Webber, 1988]), where the corresponding data structures are called quadtree and octree in two- and three-dimensional spaces correspondingly. Both of the above data structures have proved to be very efficient computationally in graphics. We extend the idea from two- and three-dimensional space partitioning to multidimensional domains and tailor the techniques for our purposes.

[1] Discrete children happen quite often in practical modeling. For example, we might need to model a thermostat switch turned on by temperature or a fire alarm sensor turned on by smoke concentration. Some other examples will be given later in this chapter.

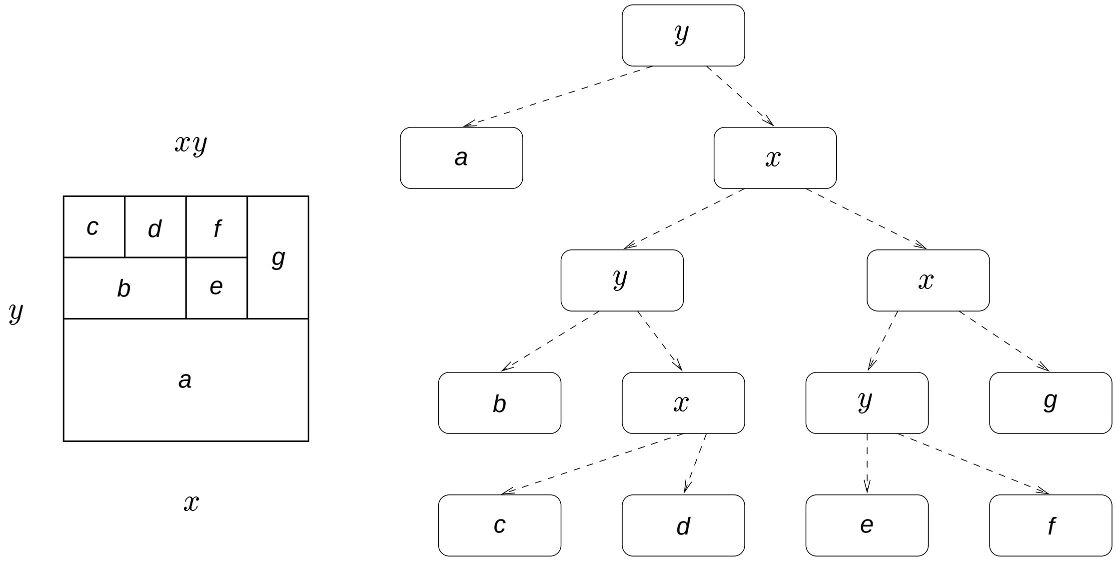


Figure 6.1: An example of a two dimensional hierarchical space decomposition. Internal nodes of the tree store the axis of a split. Leaves of the tree store the average of the continuous density function over the subregion represented by the leaf.

In a BSP tree a node represents a subspace of the original space. The children of a node represent a decomposition of the original subspace into smaller subspaces. In a BSP tree, we restrict the decompositions to subdivisions of the space into two halves by a plane orthogonal to one of the axes. For example, Fig. 6.1 shows one possible BSP tree for a function $f(x, y)$ of two variables x and y. On the first level, we split the function domain by a line orthogonal to y. Each of the resulting states represents a half of the original state space $[0 \leq y \leq 0.5]$ and $[0.5 \leq y \leq 1]$. On the second level, we leave the left node as a leaf representing the abstracted state $[0 \leq y \leq 0.5]$, the lower half of the xy plane. We split the right one, representing the abstracted state $[0.5 \leq y \leq 1]$ (the upper half of the xy plane) by a line orthogonal to x. Each of the children on the third level is split even further. The splitting continues from the root of the tree, representing the whole function domain $\Omega = \{x, y\}$, to the leaves, that carry information about the abstracted function $f_{\mathcal{A}}(x, y)$ in a particular subregion ω_i.

A leaf of the tree stores the clique specific information: the value of the abstracted function $f_{\mathcal{A}}(\omega_i)$ and the weight $w(\omega_i)$, both of which are constant in the subregion ω_i defined by the leaf. BSP trees can be readily used for probabilistic inference. A

BSP tree is closed under operations for probabilistic inference: summation, multiplication, and integration (the algorithms for doing these operations are described in Appendix B). Thus probabilistic inference—local clique computations and message passing—can be carried out in terms of BSP trees, i.e., all intermediate results can be represented as BSP trees.

6.1.2 Partitioning algorithm

Fig. 6.2 describes our BSP tree construction algorithm, which is a greedy algorithm trying to minimize the total WKL distance. The leaves of the BSP tree are kept in a priority queue based on the estimate of WKL distance contribution from this leaf. The leaf with the largest WKL distance is split first, and the two resulting leaves are put back into the queue. Correspondingly, the leaf contribution to the total WKL distance is replaced by the sum of the two contributions from the two leaves resulting from the split.

```
place a BSP tree node representing the whole region Ω in a priority queue
repeat
   take a BSP node out of the priority queue
   find the optimal split direction for the given BSP node
   form two new BSP tree nodes and place them in the priority queue
   estimate the change in the total WKL distance for the BSP tree
   estimate the new WKL distance for the BSP tree abstraction
until the tree size is larger than M or the precision of the cumulative BSP tree
abstraction error is less than δ
return the resulting BSP tree structure, reestimating the abstracted potentials and
WKL errors in the leaves
```

Figure 6.2: Partitioning algorithm.

First, we need to have a method to estimate the contribution of each leaf to the total WKL distance in order to prioritize the leaves in the priority queue. To estimate the error we would ideally perform a multidimensional integration over the leaf subregion ω_i (since the weights are constant in the leaves, the WKL distance computed over a leaf is also the KL distance over the same leaf multiplied by the constant weight). Multidimensional integration is not acceptable since it is very expensive computationally. Thus, we resort to an estimation of the WKL distance based on the function f mean $\bar{f}$, which is equal to $f_{\mathcal{A}}$, the function maximum f_{max}, and the function minimum f_{min} in the given subregion ω_i:

$$w\int_{\omega_i} f\log\frac{f}{\bar{f}}\,d\Omega \le w\Big[\frac{f_{max}-\bar{f}}{f_{max}-f_{min}}f_{min}\log\frac{f_{min}}{\bar{f}}+\frac{\bar{f}-f_{min}}{f_{max}-f_{min}}f_{max}\log\frac{f_{max}}{\bar{f}}\Big]|\omega_i|, \tag{6.3}$$

where $|\omega_i|$ denotes the volume of a subregion ω_i. We derive this bound in Appendix C. The parameters $\bar{f}$, f_{max}, f_{min} are estimated by randomly sampling f at several points.

To estimate the direction of the optimal split we would optimally try to estimate the WKL metric gains due to each possible split, which is also expensive. Instead, we draw a line parallel to one of the axes through the center of Ω and sample several points on this line. We choose a split plane to be orthogonal to the direction along which the function changes most within Ω, i.e., the ratio f_{max}/f_{min} along the line is maximum among all possible directions. For the sampling, we use 10 equidistant points per direction.

After the structure of the BSP tree is fully determined, the new clique potential—the average of the function over the leaf subregion—is estimated by a Monte Carlo integration technique. While we used 16 function evaluations per leaf for estimating parameters $\bar{f}$, f_{max}, and f_{min} (see (6.3)), we used 128 function evaluations per leaf for estimating the final averaged clique potential and the final leaf contribution to the WKL distance. The weight for the two children was set to the weight of the parent.

The result for the one-dimensional Ω abstraction of a normal probability distribution (1.3) with $\mu = 0.5$ and $\sigma = 0.05$ and a constant weight is shown in Fig. 6.3. We also show the "optimal" abstraction in which we do not limit the location of the splits and which was found by a gradient descent method to minimize the KL distance. As we can see, the abstraction we obtained with a BSP tree is very close to the "optimal" abstraction.

Fig. 6.4 shows the KL distance as a function of the number of subregions for the BSP tree and "optimal" abstraction obtained with a gradient descent method of a

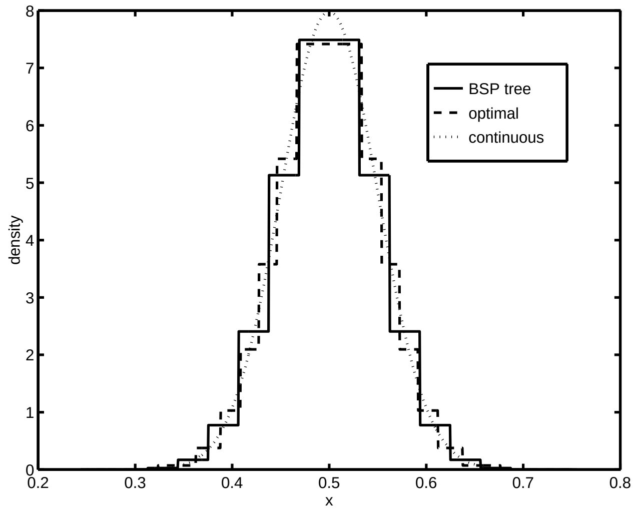


Figure 6.3: BSP tree (solid line) and "optimal" (dashed line) discretization of a normal distribution $N(x; 0.5, 0.0025)$ (dotted line). The number of abstraction subregions is 16 in both cases. The "optimal" abstraction was found by the gradient descent method.

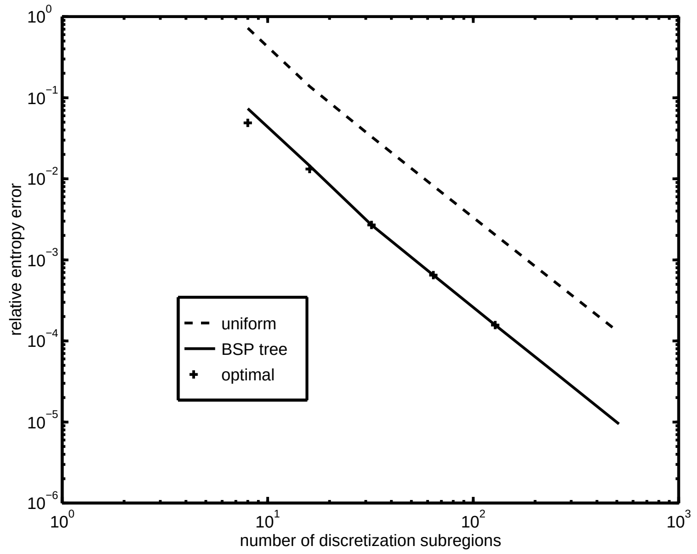


Figure 6.4: Relative entropy error of abstraction as a function of the number of subregions for equidistant (dashed line), BSP tree (solid line), and gradient descent (crosses) abstraction. The error of the BSP tree and gradient descent abstraction are almost identical for large number of subregions. The error of the equidistant abstraction is about a factor of ten larger.

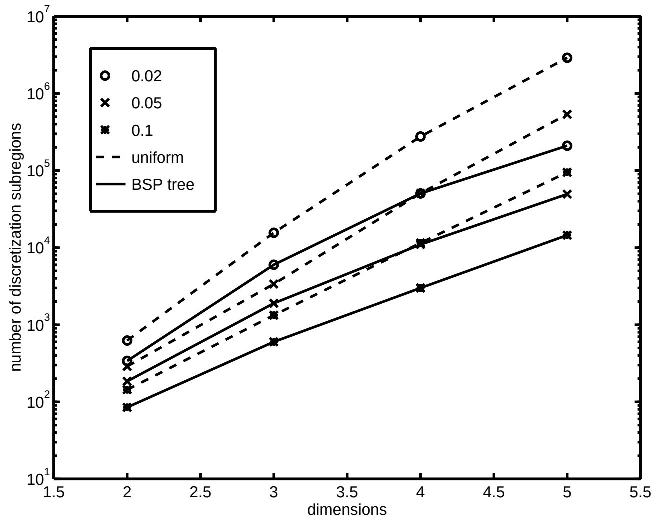


Figure 6.5: Number of subregions in an abstracted function as a function of the dimensionality for BSP tree (solid line) and uniform (dashed line) abstraction given the same precision measured by KL distance. The abstraction was performed to approximate a multivariate normal distribution proportional to $N(\sum_1^{n-1} x_i/(n-1) - x_n; 0, 0.0025)$ with fixed KL distances of 0.02, 0.05, and 0.1. For a large number of dimensions, the BSP abstraction performs much better (notice the logarithmic scale for the number of subregions).

normal distribution. The resulting KL distance for these two abstractions is virtually the same for the number of intervals larger than 16. On the other hand, an equidistant abstraction corresponding to an equidistant subdivision of the interval $[0, 1]$ requires about a factor of 5 more splits to reach the same accuracy.

Fig. 6.5 shows the dependence of the number of subregions to abstract a function (for a fixed precision) on the function domain dimensionality. Abstraction with BSP tree might require exponentially fewer subregions since we save a constant factor in each dimension (see Fig. 6.4). For a given accuracy, the number of subregions grows much slower for a BSP tree abstraction than for a uniform abstraction. We save about a factor of 10 in 5 dimensions. The shape of this function, a ridge along the hypercube diagonal, was much harder to fit with a BSP tree in the multidimensional case than in the one-dimensional.

6.1.3 Iterative inference algorithm

To efficiently bound the final error of probabilistic inference we have to minimize the WKL distance in each clique:

$$W\left(f(C_k) \| f_{\mathcal{A}}(C_k); w(C_k)\right), \tag{6.4}$$

where the weights are the posterior divided by the prior (see (4.13)). To get the correct posterior we have to have the result of probabilistic inference which is clearly not available to us to begin with.

```
build a join tree (factoring) for the BN graph
assign continuous functions to cliques that completely contain all their arguments
assign a guess about the weight (uniform weight) to all cliques
find a clique that contains the query node and make it the root of the tree
repeat
  for each clique starting from the leaves and up to the root do
    multiply all messages from descendants
    multiply the previous result by the assigned functions
    build a BSP tree abstraction of the resulting product
    form a message up by marginalizing over variables that do not appear in the
    parent
  end for
  for each clique starting from the root and down to the leaves do
    calibrate the product of the weight and potential to the message from the
    parent, if any
    form messages down by marginalizing over variables that do not appear in a
    child
  end for
until the posterior probability density converges
```

Figure 6.6: An iterative BSP tree algorithm for hybrid networks.

We resolve this circularity problem by an iterative algorithm shown in Fig. 6.6. First, we assign a constant weight, for example one, to all cliques. Then we iterate over up and down propagations. On the way up we compute an approximate posterior for the root clique (compare to the up propagation in Fig. 2.3). On the way down, we propagate the information about the posterior down the tree.

The only substantial difference between the down propagation in the iterative algorithm and the algorithm in Fig. 2.4 is that we adjust the *weight* rather than the *potential* in each clique, so that the product of the weight and potential is calibrated. We multiply the weight in the clique to be calibrated by the ratio of the old and the new message on the edge $\langle k, l \rangle$:

$$w'(C_l) = \frac{m'(C_k \cap C_l)}{m(C_k \cap C_l)} \times w(C_k), \tag{6.5}$$

where the new message $m'(C_k \cap C_l)$ is computed by integrating the product of potential and weight of the clique C_k:

$$m'(C_k \cap C_l) = \int_{C_k \setminus (C_k \cap C_l)} f(C_k)\, w(C_k)\; d\Omega \tag{6.6}$$

(compare to (2.17)).

6.2 Networks

As we described in the introduction, a hybrid network is a BN that contains continuous variables. One of the simplest examples of a hybrid network was shown in Fig. 1.2. Let us present more practical and complex hybrid network examples.

6.2.1 Object monitoring

Let us assume that we can make discrete observations of a one-dimensional continuous robot position on an interval from zero to one. The observations are noisy—we will describe the specific conditional probabilities below—and the robot moves randomly between the observations. The hybrid BN corresponding to this problem is shown in Fig. 6.7.

The first two observations, variables o_1 and o_2, are noisy observations of the robot position. If the robot position is x, the first and the second sensor readings are o with

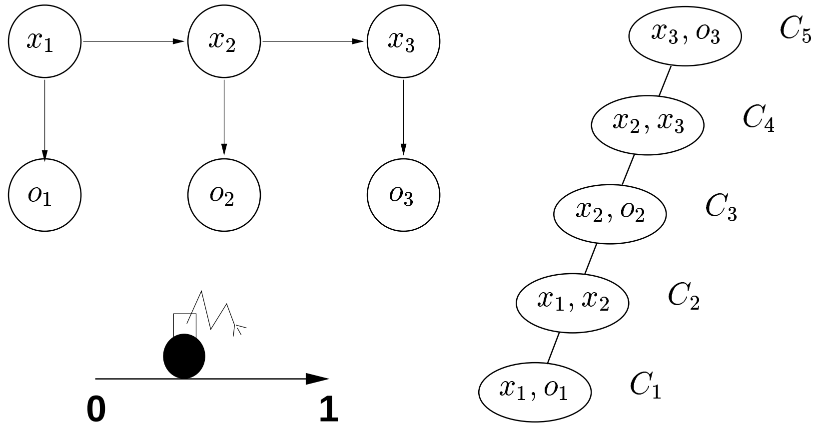


Figure 6.7: A simple hybrid BN and its join tree.

probability:

$$p(o|x) = N(o - x; 0, 0.01), \tag{6.7}$$

where $N(x; \mu, \sigma^2)$ is a normal density function (1.3). The third observation is a noisy observation of the robot in the left half-space $x < 0.5$. If the robot position is x, the sensor is likely to give a reading o of *true* with probability:

$$\frac{1}{1 + \exp(40(x - 0.5))}. \tag{6.8}$$

Note that the last observation is a discrete variable. The dependence between successive time slices was a normal probability distribution $p(x_n|x_{n-1}) = N(x_n - x_{n-1}; 0, 0.01)$.

Although we do not know the robot's position at the beginning of the observation chain, we might have a good idea where the robot is after a sequence of observations. However, the importance of the first observation is smeared out since the robot might have traveled between the observations. The problem is to find the probability distribution over the values of the robot position, i.e., the conditional probability $p(x_3|o_1, o_2, o_3)$ of the robot position x_3 at the last step in the chain given all three observations o_1, o_2, and o_3.

While the answer to a query in discrete only networks can be found by a summation over joint probability states, the answer to a query in a hybrid network can be found by a multidimensional integration. A join tree for the above network is shown in Fig. 6.7 and corresponds to the following query factoring:

$$\begin{aligned}
p(x_3|o_1, o_2, o_3) &\equiv p(x_3, o_1, o_2, o_3)/p(o_1, o_2, o_3) \\
&= \alpha \times \int_0^1 \int_0^1 p(o_3|x_3)p(x_3|x_2)p(o_2|x_2)p(x_2|x_1)p(o_1|x_1)p(x_1)\, dx_1\, dx_2 \qquad (6.9) \\
&= \alpha \times p(o_3|x_3) \int_0^1 p(x_3|x_2)p(o_2|x_2) \left(\int_0^1 p(x_2|x_1)p(o_1|x_1)p(x_1)\, dx_1 \right) dx_2.
\end{aligned}$$

We picked the example so as to allow us to compare the results of our program to the exact analytical answers. The exact analytical solution for the observations o_1, o_2, $o_3 = \mathit{false}$ is:

$$p(x_3) \sim \frac{N(x_3; (o_1 + 2o_2)/3, 1/60)}{1 + \exp(-40(x_3 - 0.5))}; \qquad (6.10)$$

and the solution for the observations o_1, o_2, $o_3 = \mathit{true}$ is:

$$p(x_3) \sim \frac{N(x_3; (o_1 + 2o_2)/3, 1/60)}{1 + \exp(40(x_3 - 0.5))}; \qquad (6.11)$$

where the answer was obtained by integrating (6.9) over x_1 and x_2.

6.2.2 Damper diagnosis

A more practical problem with which we have experimented, is diagnosis of air duct dampers.[2] A damper is a part of commercially available damper box, which regulates air flow in a building. Almost any commercial building contains at least one damper box. The location of the damper boxes makes it often difficult to directly observe the damper functioning.

Thus, we need a method to be able to derive damper internal status variables

[2]The model is the courtesy of Robert Dodier of University of Colorado at Boulder and represents a real practical problem he was working on. We highly appreciate his cooperation in providing us with the data.

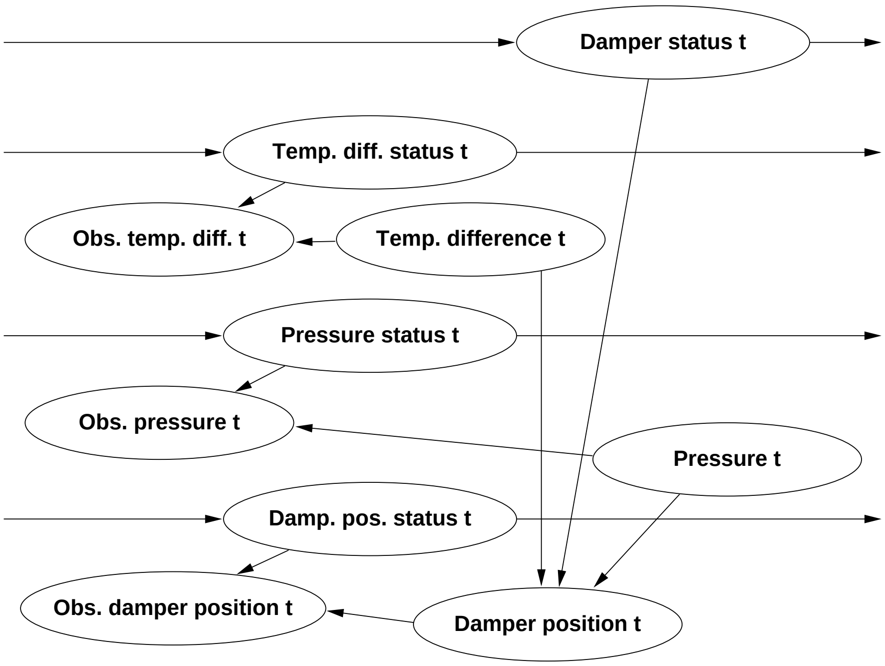


Figure 6.8: A damper diagnosis BN.

through the external observations. An instance of a typical damper diagnostic problem is shown in Fig. 6.8. The figure shows one time slice out of a sequence of damper observations. The observations are: "Observed temperature difference t"—the difference between the desired temperature and the actual room temperature; "Observed pressure t"—pressure in the air duct; and "Observed damper position t"—the quantitative measure of how much the damper is open. We abbreviate the observed nodes as x_{OTD}, x_{OP}, and x_{ODP} and the corresponding hidden variables as x_{TD}, x_{P}, and x_{DP} correspondingly.

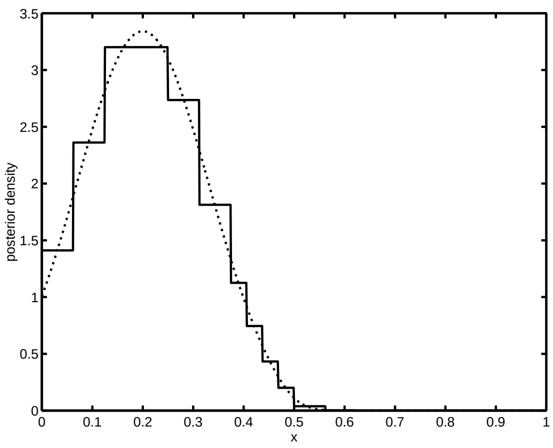
The observations x_{OTD}, x_{OP}, and x_{ODP} are made at a discrete time intervals; given a sequence of these observations, we would like to infer the probability of four internal status variables: "Damper status t", describing the working mode of the damper; "Temperature difference status t", describing the condition of the temperature difference sensor status; "Pressure status t", describing the condition of the pressure sensor; and "Damper position status t", describing the status of the damper position sensor. We abbreviate the status nodes as x_{DS}, x_{TDS}, x_{PS}, and x_{DPS} correspondingly.

The specific instantiations of the conditional probabilities we used for this problem can be found in Appendix D. Although none of the observations is discrete, the dependence $p(x_{\mathrm{DP}}|x_{\mathrm{TD}})$ of damper position on temperature difference is a stepwise dependence and thus cannot be modeled by a gaussian function. Given all three observations, the time slice represents a clique with 4 discrete and 3 continuous variables, which makes it quite difficult to integrate with traditional methods.

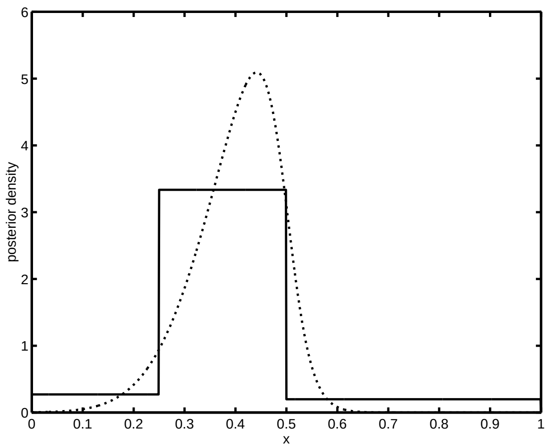
6.3 Results

We tested our algorithm on the two problems described above. The object monitoring problem (see Section 6.2.1) has the exact solution (6.10) and (6.11). The damper diagnosis problem (see Section 6.2.2) can be solved exactly only by multidimensional integration and we provide the comparison of the two techniques.

6.3.1 Object monitoring



(a) $o_1 = o_2 = 0.2$



(b) $o_1 = 0.2$, $o_2 = 0.65$

Figure 6.9: Posterior probability $p(x_3)$ for a network shown in Fig. 6.7 for similar (a) and contradictory (b) evidence. The results of the inference are shown by a solid line, the exact result is shown by a dotted line. Evidence o_3 is *true* in both cases.

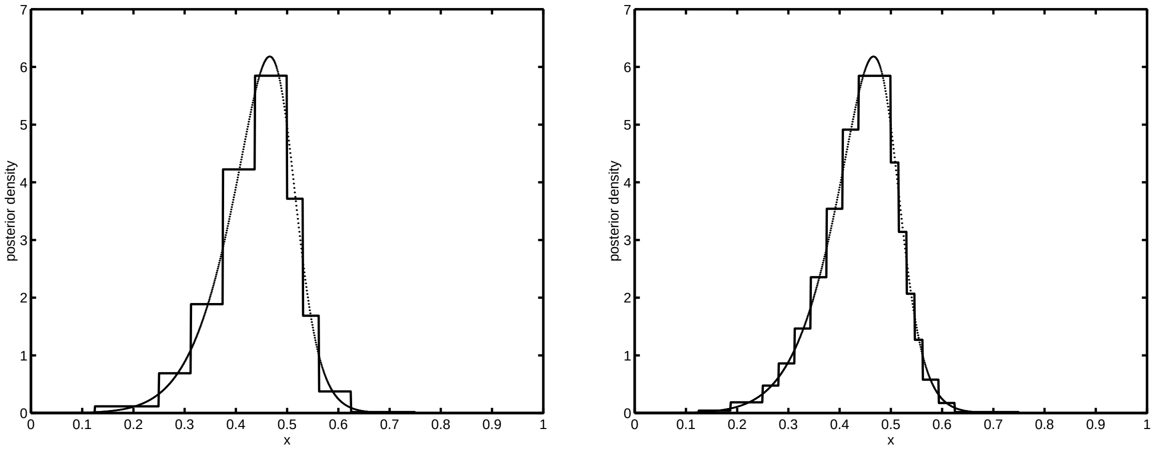
First, we checked how the algorithm performs on the first iteration. In many cases, even the first iteration produced good enough results. For example, as we see in Fig. 6.9, if our first two observations are the same—$o_1 = o_2 = 0.2$—the results of the inference are very close to the exact solution, computed analytically in (6.11). However, as our observations become more and more unlikely, the accuracy of the results of the first iteration begins to deteriorate. If, for example, the second observation is changed to $o_2 = 0.65$, the results of the inference contain only a single bump (see Fig. 6.9) and are very different from the exact answer.

Weight reassignment

The poor results for unlikely evidence are easy to understand: the greedy BSP construction algorithm spends too much time refining the regions which do not contribute much to the precision of the final answer. To make the algorithm work well with the unlikely evidence, we have to guide the discretization program and to force it to discretize the regions which are more important for the final precision.

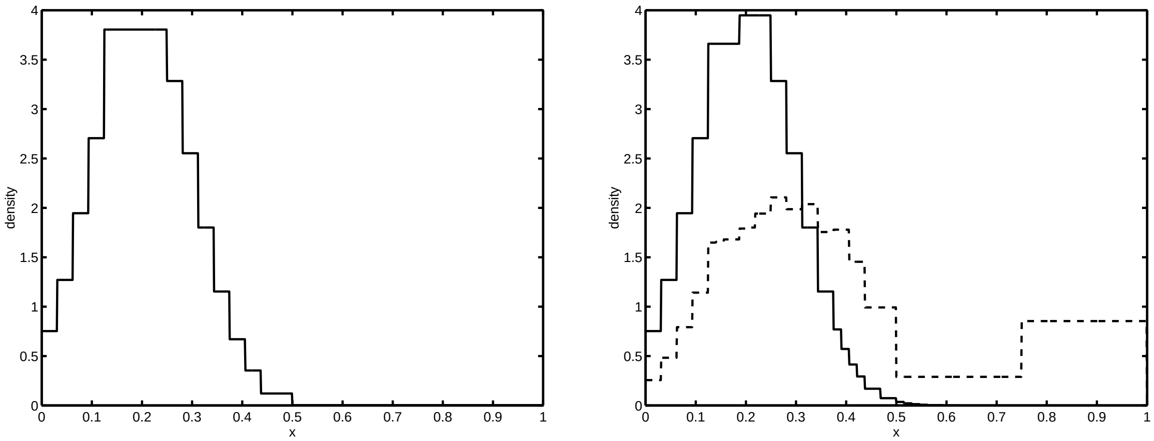
In the next experiment we used a very unlikely evidence $o_1 = 0.2$, $o_2 = 0.8$, and $o_3 = \mathit{true}$, which resulted in the BSP tree with only one leaf on the first iteration, reflecting a very poor initial abstraction granularity which was done with uniform weights. Already on the second iteration, after only one phase of weight propagation, the cliques had a very good estimate of the posterior distribution and therefore the weights. The BSP tree on the second iteration had 11 leaves, and the posterior probability distribution differed from the true probability distribution by a KL distance of 0.03. The BSP tree after the third round of propagation had 20 leaves, and the posterior probability distribution differed from the true probability distribution by a KL distance of 0.01 (see Fig. 6.10).

The BSP tree splits before and after the first weight update are shown in Fig. 6.11. While at the initial propagation the $N(x_1; 0.2, 0.01)$ multivariate normal distribution in the clique $C_1 = \{o_1, x_1\}$ corresponding to the product $p(o_1|x_1)p(x_1)$ is abstracted with uniform weight, the weight is substantially nonuniform for the second round of propagation as shown in Fig. 6.11(b). The new BSP tree structure takes into account much larger weights on the right slope and rediscretizes it more finely.



(a) Second iteration (b) Third iteration

Figure 6.10: Posterior probability $p(x_3)$ for a network shown in Fig. 6.7 with the dynamic algorithm for two successive iterations. The result of the inference is shown by a solid line, the exact result is shown by a dotted line. Evidence is $o_1 = 0.2$, $o_2 = 0.8$, and $o_3 = true$.



(a) Prior probability $p(x_1)$ (b) Rediscretized prior $p(x_1)$

Figure 6.11: Original and rediscretized prior probabilities $p(x_1)$. The dashed line shows the estimate of the posterior over x_1 that the clique has after the first weight propagation. Notice the change in the granularity of the discretization. Evidence is $o_1 = 0.2$, $o_2 = 0.8$, and $o_3 = true$.

Convergence

We provide only empirical convergence results here. The algorithm converged by the third iteration for our first problem. Figure 6.12 shows the KL distance of the final answer $p'(x_3)$ as a function of the iteration number for several goal precision parameters δ. The KL distance dropped very abruptly after the first iteration and again after the second, after which it experienced small oscillations up and down around the final answer.

Let us compare the efficiency of our approach to the integration on a uniform grid. Since we can effectively focus on the most important parts of the problem, we expect to get a better precision given the same number of function exp() evaluations. Our abstraction method gets a factor of 4 better precision as compared to the case of standard integration as shown in Fig. 6.13, which compares the relative entropy error of our algorithm compared to a standard integration on a uniform grid given the same number of function evaluations. In practice, the savings grow with the required precision and the dimensionality of the clique state space.

6.3.2 Damper diagnosis

Let us show the results for a more complex damper problem which includes cliques of dimension seven (three of which are continuous). We constructed a test problem of four time slices with observations of all three variables in the second and the third time slices. In each of the two time slices we observed temperature difference, pressure, and damper position. We converged to the goal precision of less than 0.005 in each clique within 6-8 iterations.

Fig. 6.14 shows the resulting probabilities for the status of different sensors to be in normal condition given observations in the first two time slices $x_{\mathrm{OTD1}} = 0.5$, $x_{\mathrm{OTD2}} = 0.5$, $x_{\mathrm{OP1}} = 0.5$, $x_{\mathrm{OP2}} = 0.5$, $x_{\mathrm{ODP1}} = o_1$, $x_{\mathrm{ODP2}} = o_2$, where we varied o_1 and o_2 on the interval $[0, 1]$. The probabilities $p(x_{\mathrm{TDS2}} = \text{“}normal\text{”})$ and $p(x_{\mathrm{PS2}} = \text{“}normal\text{”})$ only slightly depend on o_1 and o_2 since the damper position observations are unlikely to tell us much about the temperature and pressure status variables. On the other hand, the observations of damper position are provide much more information about

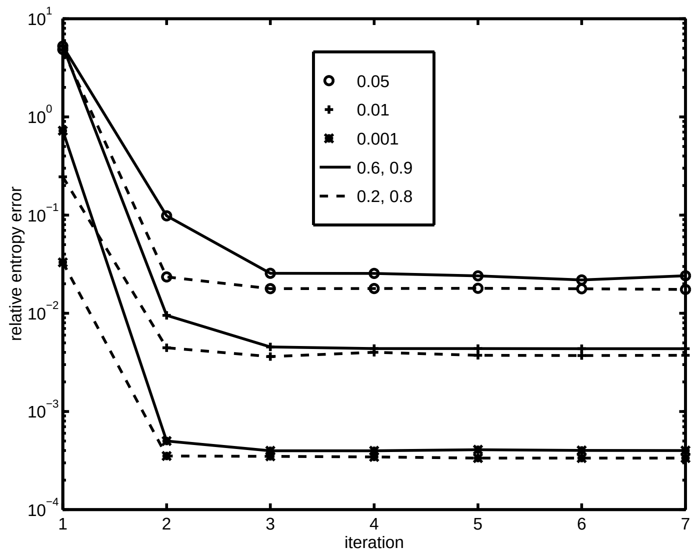


Figure 6.12: Relative entropy error as a function of the iteration number and the precision parameter δ. Evidence is $o_1 = 0.6$, $o_2 = 0.9$ (solid line) and $o_1 = 0.2$, $o_2 = 0.8$ (dashed line). o_3 is always *true*.

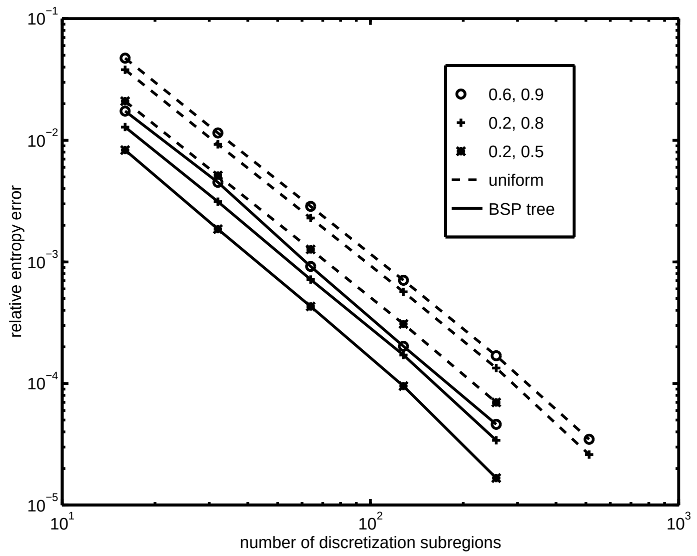
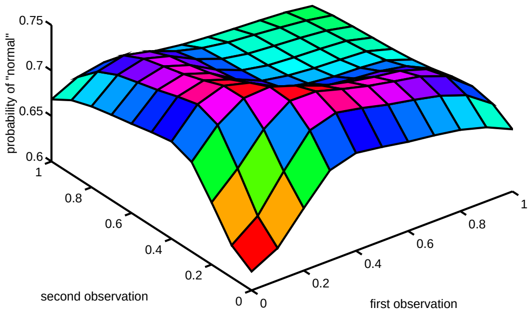
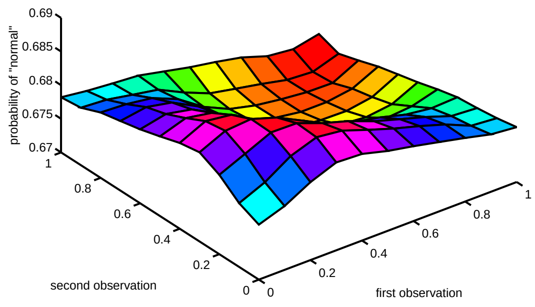


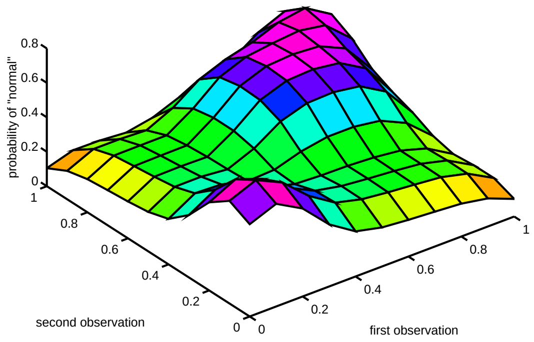
Figure 6.13: Relative entropy error as a function of the number of subregions for abstraction and evidence. Evidence is $o_1 = 0.6$, $o_2 = 0.9$ (circles), $o_1 = 0.2$, $o_2 = 0.8$ (pluses), and $o_1 = 0.2$, $o_2 = 0.5$ (stars). o_3 is always *true*.



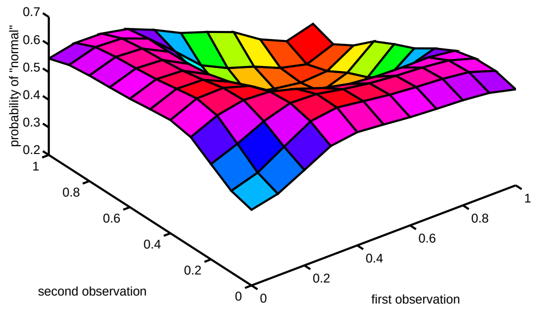
(a) x_{TDS} – temperature difference status



(b) x_{PDS} – pressure difference status

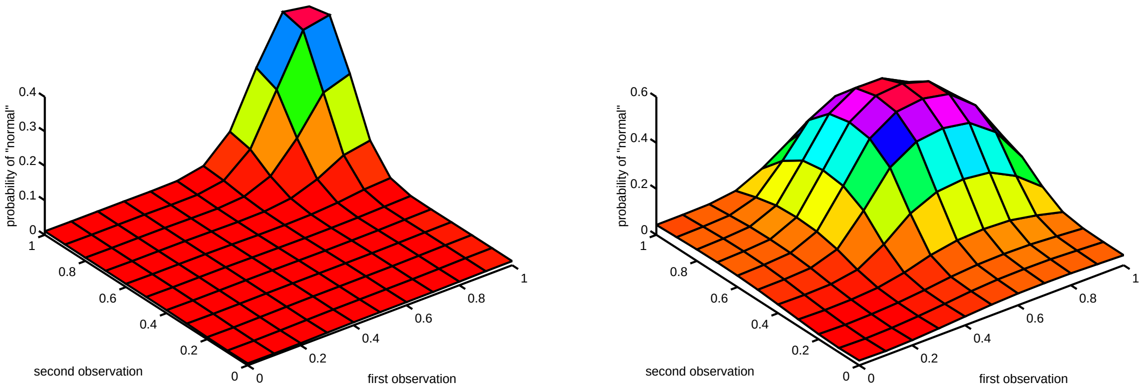


(c) x_{DPS} – damper position status



(d) x_{DS} – damper status

Figure 6.14: Probability of different sensors working correctly for two subsequent observations of the damper position. The temperature difference and pressure difference observations are fixed and are 0.5.



(a) *"stuck open"* (b) *"stuck in between"*

Figure 6.15: Probabilities of the damper status to be *"stuck open"* and *"stuck in between"*. The two probabilities compete around $o_1 = o_2 = 0.95$ causing a slow convergence of both the direct integration and the dynamic abstraction algorithms.

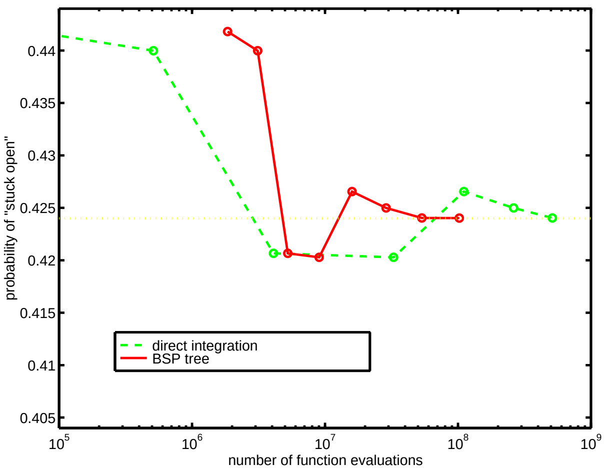


Figure 6.16: Convergence of the probability of the damper status to be *"stuck open"* for integration on a uniform grid (dashed line) and abstraction (solid line). Abstraction outperforms direct integration by an order of magnitude. Horizontal axis shows the number of threshold function evaluations (see text).

the damper status and damper position status variables.

Let us examine the efficiency of our algorithm. We compare the amount of work required to achieve a certain precision with two methods: our abstraction and direct integration on a uniform grid. We plot the achieved results for the probability $p(x_{\mathrm{DS}} =$ *"stuck open"*) on the number of threshold function:

$$f(x) = p(x_{\mathrm{DP}}|x_{\mathrm{TD}}, x_{\mathrm{DS}}, x_{\mathrm{P}}), \tag{6.12}$$

which was the most expensive part of the evaluation, for $o_1 = o_2 = 0.95$, where the convergence seemed to be particularly slow. We found that our abstraction technique requires about an order of magnitude less work than the standard integration technique (see Fig. 6.16). For example, to reach the final precision of .005 our algorithm required only about 10^7 function evaluations, while the standard integration on a uniform grid required 10^8 function evaluations.

6.4 Related work

The most popular algorithm for inference in hybrid networks remains the one based on CG networks [Lauritzen and Wermuth, 1989; Lauritzen, 1992]. It has two deficiencies: the dependencies in CG networks are very restrictive (gaussian) and one is not able to model discrete children of a continuous variable (a discrete sensor triggered by a continuous parameter).

The authors in [Driver and Morrel, 1995; Alag and Agogino, 1996] try to deal with the latter deficiency by representing an arbitrary function between two continuous variables as a mixture of CG functions. If this is done, probabilistic inference is possible, but is limited to very small network sizes. The decompositions often involve many terms and the number of terms in the sums increases exponentially with the propagation length. The only way to deal with the second deficiency, the discrete child of a continuous node, is to discretize all ancestors of the discrete variable.

In connection with our technique, we also have to mention the numerical integration on an adjustable grid used in many fields including hydrodynamics. The BSP tree integration is equivalent to integration of a function on an adjustable multidimensional grid. Thus, our construction is a way to target the grid structure to our goals based on the results of previous less detailed computations.

6.5 Conclusions

Continuous variables are practically important since many real-world variables such as temperature, pressure, fluid level, velocity, location and others are continuous. In practice, continuous variables are typically discretized before probabilistic inference. Unfortunately, any discretization lacks some of the information in the original dependencies.

Any discretization into fixed intervals cannot completely account for the future evidence and the demands of the future task(s). In particular, if we had to guarantee the same uniform precision over a wide range of evidence cases, we would have to discretize each variable into many fine intervals. The model would quickly become very large and computationally intractable.

We developed a new discretization algorithm based on our abstraction approach. The algorithm dynamically adjusts the abstraction structure to the required task and precision. In other words, if we get an evidence that was improbable, we are able to refine the abstraction granularity that delivers the result of high quality in a limited amount of time.

Our approach results in the first practical algorithm for hybrid networks with arbitrary dependence between nodes in arbitrary topology. Previous algorithms could deal only with either a CG network or a mixture of CG networks. In particular, they could not solve a network where continuous variables have discrete children, which is possible in our algorithm.

The internode dependencies can be completely arbitrary as long as we can provide a computable function that can be computed during the runtime (even one written in C++). The algorithm performs evaluations of this function as often as it considers necessary to obtain the required abstraction and precision. To optimize the abstraction and thus to minimize the number of function calls, it uses our information-based

WKL distance metric.

We expect the abstraction approach to be significant in problems where both evidence and required precision vary widely in different circumstances. The algorithm can efficiently adjust to the new requirements without much computation or memory overhead. The abstraction structure can be efficiently stored in memory and reused. Moreover, it adapts to different circumstances, placing emphasis on right regions, unlike the straightforward integration.

As a result, we can get large computation time savings. For example, to compute one point in Fig. 6.14 with the standard technique (to achieve required precision of 0.005), we will need to perform about 10^8 function (6.12) evaluations, which translates to about one day of computation on an 200 MHz R10000 processor. To achieve the same precision with our technique, we need to perform about an order of magnitude less function evaluations. In fact, we computed the whole set of points for the plots 6.14 - 6.15 within a week.

Chapter 7

Conclusions

7.1 Summary

Probabilistic inference in BNs is *NP*-hard and computationally expensive [Cooper, 1990; Dagum and Luby, 1993]. Thus, it is important to develop techniques to speed up probabilistic inference. In this thesis, we tried to address this problem from two directions. First, we explored the idea of parallelizing the exact inference algorithm. Second, we studied the possibilities of speeding up probabilistic inference by doing approximations. Although interrelated, these two approaches focus on completely different aspects of probabilistic inference.

Parallelism is a powerful technique to speed up a computation by using several processors. When parallelizing an application, we have to divide the computation into a set of almost independent computational tasks. The issues here are load balance and data locality. Load balance characterizes how evenly the processors are loaded with work so that one does not have to wait for another. Load balance often competes with data locality, i.e., how much communication these subtasks need. The independent subtasks are often of different size and do not satisfy load balance. Thus, we need to make compromises.

In this thesis we show that the tradeoff between load balance and data locality is sharpened in probabilistic inference. The number of operations for probabilistic inference is roughly proportional to the number of memory accesses with a small constant, unlike other traditional scientific applications for parallel processing. This property leads to the intensity of data accesses during a probabilistic inference computation. We show that techniques to preserve data locality are important for obtaining good performance in a uniprocessor as well as in a multiprocessor implementation.

Specifically, in Chapter 3 we show that we might sacrifice some forms of load balance to achieve better data locality. One of our implementations that uses only one form of concurrency, i.e., only in-clique but not the topological parallelism, produces better speedups than an implementation that uses both types of parallelism but sacrifices some data locality. Data access pattern of our programs was carefully analyzed with on-chip performance counters also showing the effects of the data intensity of probabilistic inference. Thus, probabilistic inference can serve as a benchmark for testing the memory architectures of new machines.

Parallelism still cannot and does not help us to solve large networks that have very large cliques. For example, to compute the full CPCS medical diagnostic network, we would need about 1 TB of memory. To deal with problems of this size we propose to use approximations in inference. Although even approximate inference is *NP*-hard, we can hope to get computational savings for reasonable error bounds on the results.

We chose a general approach to approximate inference based on state space abstraction. We proposed an approach where we abstract states at the clique level. The number of states at the clique level directly affects the probabilistic inference time and abstraction at the clique level is most beneficial for probabilistic inference. We describe a metric, the WKL distance, to choose the best approximation, which is based on relative entropy or KL distance. The optimal weights for the WKL distance require knowledge of the posterior distribution, which is not known up front and can be obtained only by probabilistic inference. We solve this problem by either trying to minimize the abstraction error for a "typical" query or by an iterative algorithm that iteratively improves its guess about the posterior and the weights. We demonstrate these techniques on two important classes of networks.

In Chapter 5 we apply the static state space partitioning (abstraction) to BN2O networks. In such networks, the nodes in the first layer are the query nodes and the nodes in the second layer are evidence nodes. We know that the prior probabilities of the nodes in the first layer are low in practical networks. This information helps us to choose a partitioning that minimizes the diagnosis error for all possible queries. The above partitioning is very close to the single (or more general k-fault) hypothesis used for such networks in the past, but is much more precise.

In Chapter 6 we extend the idea of static partitioning to a hierarchical dynamic partitioning, an abstraction which adjusts itself with the inference task and precision. In this approach, we do not have to constrain ourselves to any assumptions about probabilities or the type of queries in a BN. We apply this technique to hybrid networks, where the presence of continuous variables makes probabilistic inference particularly hard due to the continuous state space and possibility of very unlikely evidence. We start with a very coarse partitioning and iteratively improve our guess about the posterior, the weights, and the partitioning. We show that the procedure converges in practice and produces good quality results. We show that hierarchical abstraction can save an exponential factor in the running time. An important byproduct of the above technique is the first general purpose algorithm for hybrid networks with arbitrary dependencies for nodes in an arbitrary topology.

7.2 Future work

In this dissertation we addressed the question of managing computational complexity of probabilistic inference. Although probabilistic inference remains a hard problem, we managed to substantially reduce computation time for several classes of practical networks. Two major approaches for doing this were parallel processing and approximation by abstraction.

There are many avenues along which the results of this dissertation might be extended. Although the data locality turned out to be relatively more important than load balance for exact probabilistic inference because of data intensity, the approximate inference algorithms are usually less data intensive. Thus, the tradeoff between load balance and data locality might be solved the other way, with the preference for better load balance.

In particular, our hierarchical abstraction structure allows the same kind of "topological" parallelism as the join tree: the computations in different branches of the BSP tree are independent. These computations involve multidimensional integration and are computationally intensive. Moreover, the workload in different branches of the BSP tree as well as in different cliques of the join tree is unpredictable, which makes the dynamic load balancing preferable over the static task partitioning we used in this thesis.

In this thesis we have shown only empirical convergence of the iterative algorithm on two examples. In the future, we would like to study the general convergence properties of hierarchical abstraction. The problem here seems to be of the "circularity" nature: to prove a bound on the result of inference we have to prove a bound on weights, which are observed from the results of inference on the previous iteration.

The approach based on approximation by abstraction can also be extended by itself. In particular, we would like to extend our dynamic approach to BN with discrete only variables. Although the general approach developed in Chapter 4, the one based on WKL distance, remains valid for discrete only networks, it is much harder to come up with an efficient state space partitioning like the BSP tree partitioning which we used for hybrid networks.

Our iterative inference algorithm in hybrid networks can be viewed as an efficient integration technique of a continuous function on a self-adjustable grid. Such integration technique can prove to be beneficial for other fields requiring multidimensional integration and we would like to apply it to other problems.

Appendix A

Notations

In this thesis, I prefer to use notations that are slightly different from the standard ones. Let me describe my notations here in detail.

A BN is a collection of random variables with dependencies between them expressed as a DAG. I denote a random variable as well as a node in the DAG by small Latin letter x with a possible subscript, say x_i, when further distinction between variables is required. The subscript corresponds to the label the node has in the figure that shows the BN.

The probability that a variable x_i is in one of its states, say state *"value"*, is denoted as $p(x_i =$ *"value"*$)$. In many cases, the variable is binary and can take only two values *false* or *true*. In such cases I use a shortcut and denote $p(x_i = \mathit{false})$ or $p(x_i = \mathit{true})$ as $p(\bar{i})$ or $p(i)$ correspondingly. In a general case, a variable, like the season variable x_S in Fig. 1.2, can take many possible values. I denote a variable with multiple values or a set of variables by a capital Latin latter X with a possible subscript when further distinction is required.

A very special set of variables is the set of clique nodes. It denotes the variables of one of the partial results in the decomposition (2.6 – 2.11). To emphasize this fact, I denote a set of clique variables as C_i. Thus, the clique potential $p(C_i)$ depends on the variable $C_i = \{x_1, x_2, \ldots, x_n\}$, and the message between cliques C_i and C_j depends on the intersection $C_i \cap C_j$, or the *separator* between the cliques C_i and C_j.

I denote the number of simple nodes in the set X_i as $N(X_i)$ and the size of the state space as $|X_i|$. For example, the state space of a set of variables is a direct product of the state spaces of the variables in the set. Thus:

$$X_i = \{x_1, x_2, \ldots, x_n\} \Rightarrow |X_i| = \prod_{i=1,n} |x_i|,$$

and the state space size of a set of variable X_i is exponential in the size of the set $N(X_i)$.

Finally, I denote a set of variable states, or a *superstate*, as $[condition]$, where *condition* can be any condition on the values of the variables. For instance a superstate with the values of x_1 between a and b is denoted as $[a \leq x_1 \leq b]$. A probability distribution or density function with superstates is called an *abstracted* probability distribution or density function and is denoted with a subscript $\mathcal{A}$, for instance $f_{\mathcal{A}}(x_1, x_2, \ldots, x_n)$.

Appendix B

Operations on BSP trees

Let us start with a binary operation $\diamond$ on two BSP trees where $\diamond$ can be either summation or multiplication in our examples. Let us call the structures of two trees *aligned* if the trees have exactly the same splits on the same levels. If the structures of the operands are aligned, the operation $\diamond$ is reduced to performing $\diamond$ at the leaf level, i.e., the values stored at the leaves are summed or multiplied together, and the total operation complexity is $O(N)$, where N is the number of leaves in a tree.

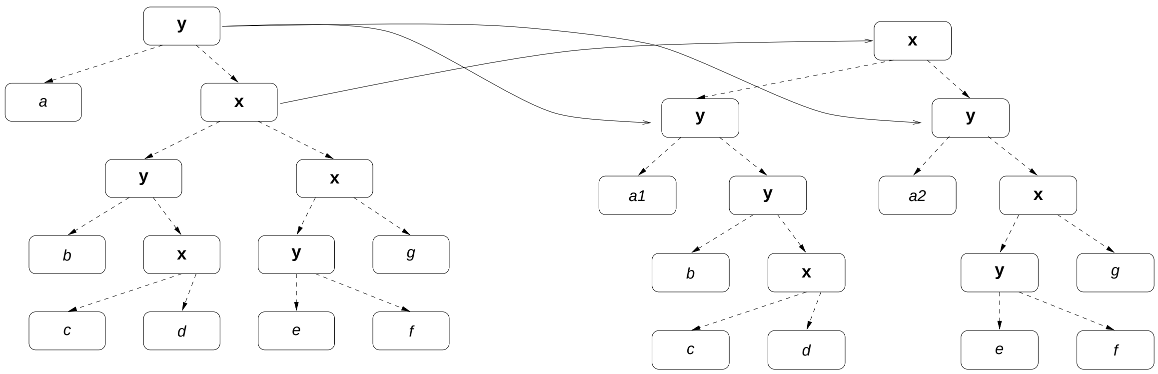


Figure B.1: Adjusting the structure of the BSP tree in Fig. 6.1 to another BSP tree that has a root split on variable x.

Now, let us look what happens if the trees are not aligned. In this case, we need to adjust the abstraction structure of one of them by inserting additional nodes. For example, if we want to sum the tree in Fig. 6.1 with another tree that has a root split on variable x, we adjust the structure of the first tree as shown in Fig. B.1 by moving the x split in the right branch up and by making two additional leaves in the corresponding branches. Complete alignment of the trees takes $O(N_1 \times N_2)$ operations in the worst case, where N_1 and N_2 are the number of leaves in the first and the second trees respectively.[1]

```
input: two nodes of a tree representing the same subregion ω_i
output: a node of a tree representing the result of the operation ◇
if both nodes are leaves then
    return a leaf with result of ◇ on the values in the leaves
else if the second node is a leaf then
    ◇ the constant value from the second node to all leaves of the first subtree
    return the first node
else
    if split on different variables then
        adjust the structure of the first tree
    end if
    ◇ left subtrees of both operands
    ◇ right subtrees of both operands
    return a node with the result of the previous two operations as its children
end if
```

Figure B.2: Algorithm for performing a binary operation $\diamond$ on two BSP trees.

The general algorithm for performing a binary operation $\diamond$ on two BSP trees is shown in Fig. B.2. The algorithm takes $O(N_1 + N_2)$ operations in the best and $O(N_1 \times N_2)$ operations in the worst case. Intuitively, if all the splits in both operands are on the same variable, the computational complexity is linear. If the trees are completely misaligned, for example if all the splits in the first tree are on variable x and in the second tree are on variable y, then the computational complexity as well as the size of the resulting tree is quadratic $O(N_1 \times N_2)$.

Let us consider integration of a function represented by a BSP tree over some variable. This is the operation that, in the discrete BN case, corresponds to variable

[1] We assume that the BSP trees are encoded as a tree structure that uses pointers. Other implementations that might be more computationally efficient are possible, but are out of the scope of this thesis. We refer the reader to [Samet and Webber, 1988] for a comprehensive review on this subject.

```
input: a node of a tree and the i-th variable to be integrated over
output: a node of the tree representing the result of the integration
if the node is a leaf then
    return the node itself
else if split on i-th variable then
    integrate left subtree
    integrate right subtree
    return the sum of the left and the right subtrees divided by two
else
    return the node itself
end if
```

Figure B.3: Algorithm for integrating a function represented as a BSP tree over a variable.

elimination by summation. Since a leaf represents a constant value of an abstracted function with a constant value in the corresponding subregion, the integration of a leaf is reduced to multiplication of the value stored in the leaf by the corresponding multidimensional volume. Since the volume of a subregion represented by a child is always half the size of the subregion represented by its parent, we can compute the subregion volume during tree traversal. The integration algorithm is presented in Fig. B.3. Integration is linear in the size of the BSP tree, i.e., it takes $O(N)$ operations.

Many other algorithms, for such tasks as computing the expected value of a function, the cross entropy, or the differential entropy, can be expressed as a simple traversal of the tree, thus taking only linear time with respect to the size of the tree.

Appendix C

Bound proof

Here we derive a bound on the KL distance integral $\int_{\omega_i} f \log f/\bar{f}\, d\Omega$ for a continuous function $f(\Omega)$ over some subregion ω_i:

$$w \int_{\omega_i} f \log \frac{f}{\bar{f}}\, d\Omega \leq w \Big[\frac{f_{max} - \bar{f}}{f_{max} - f_{min}} f_{min} \log \frac{f_{min}}{\bar{f}} + \frac{\bar{f} - f_{min}}{f_{max} - f_{min}} f_{max} \log \frac{f_{max}}{\bar{f}} \Big] |\omega_i|,$$

that we used in Section 6.1.2. The parameters $\bar{f}$, f_{max}, and f_{min} are the function average, maximum, and minimum correspondingly.

Let us divide ω_i into small subvolumes Δ so that the function $f(\Delta)$ is approximately constant in each of them. Now, let us construct function g in the following way: g equals to f_{min} in $(f_{max} - \bar{f}(\Delta))/(f_{max} - f_{min})$ part of Δ and equals f_{max} in the rest of the Δ. The average of the function g in Δ is equals $f(\Delta)$:

$$\int_{\Delta} g\, d\Omega = \frac{\bar{f}(\Delta) - f_{min}}{f_{max} - f_{min}} f_{max} + \frac{f_{max} - \bar{f}(\Delta)}{f_{max} - f_{min}} f_{min} =$$

$$\frac{\bar{f}(\Delta)\, f_{max} - f_{min} f_{max} + f_{max} f_{min} - \bar{f}(\Delta)\, f_{min}}{f_{max} - f_{min}} = \bar{f}(\Delta)\, \frac{f_{max} - f_{min}}{f_{max} - f_{min}} = \bar{f}(\Delta).$$

Since the function $x \log x$ is concave, the contribution from Δ to the full integral of $f \log f$ over ω_i is always smaller than the contribution from Δ to the full integral of

$g \log g$ over ω_i. Since we can make the subvolumes Δ arbitrary small (thus the errors due to the fact that f is not actually constant in Δ can be made arbitrary small) and all contributions from the subvolumes Δ are smaller for $f \log f$, the full integral $\int_{\omega_i} f \log f \, d\Omega$ has to be smaller than the integral $\int_{\omega_i} g \log g \, d\Omega$. The two integrals are the same if and only if $f(\Omega)$ is constant.

Note that the averages over ω_i for both functions are equal, i.e., $\bar{f} = \int_{\omega_i} f \, d\Omega / |\omega_i| = \int_{\omega_i} g \, d\Omega / |\omega_i| = \bar{g}$, since the averages are equal in each Δ. Given the above, we have:

$$\int_{\omega_i} f \log \frac{f}{\bar{f}} \, d\Omega = \int_{\omega_i} f \log f \, d\Omega - \int_{\omega_i} f \log \bar{f} \, d\Omega =$$
$$\int_{\omega_i} f \log f \, d\Omega - \int_{\omega_i} g \log \bar{f} \, d\Omega \leq \int_{\omega_i} g \log g \, d\Omega - \int_{\omega_i} g \log \bar{f} \, d\Omega =$$
$$\int_{\omega_i} g \log \frac{g}{\bar{f}} \, d\Omega = \left[\frac{f_{max} - \bar{f}}{f_{max} - f_{min}} f_{min} \log \frac{f_{min}}{\bar{f}} + \frac{\bar{f} - f_{min}}{f_{max} - f_{min}} f_{max} \log \frac{f_{max}}{\bar{f}} \right] |\omega_i|,$$

which is the same bound we gave in (6.3). In the last equation we used the fact that $g(\Omega)$ is equal to f_{min} in $(f_{max} - \bar{f})/(f_{max} - f_{min})$ part of the ω_i and to f_{max} in $(\bar{f} - f_{min})/(f_{max} - f_{min})$ part of the ω_i (otherwise, the functions f and g do not have the same average overall).

Appendix D

Damper problem conditional probabilities

The cardinality of the variables and their stationary probability distributions, which are also the distributions in the first time slice, are given by Table D.1. The variable "Damper status" has four possible values, all other status variables have two possible values. The variables "Temperature difference", "Pressure", "Damper position", "Observed temperature difference", "Observed pressure", and "Observed damper position" are continuous.

The dependence of the variable "Damper status $t(i)$" on the variable "Damper status $t(i-1)$" is given by Table D.2 where the parameter a was 0.005 in our experiments. The dependence of the rest of the status variables is given by Table D.3 where the parameter b was also 0.005 in our experiments.

The "Temperature difference" and "Pressure" variables in different time slices are independent (in the absence of the sensor information). The sensor conditional probabilities are given by the Table D.4 where σ_{OTD}, σ_{OP}, σ_{ODP}, σ_0 were 0.1 and c was 0.5 in our experiments.

Finally, the dependence of the damper position on the temperature difference and pressure for the *normal* damper status is given by:

$$p(x_{\mathrm{DP}}|x_{\mathrm{TD}}, x_{\mathrm{P}}, x_{\mathrm{DS}} = normal) = N(x_{\mathrm{DP}} - F(x_{\mathrm{TD}}, x_{\mathrm{P}})|0, \sigma^2_{\mathrm{DP}})$$

name	values	initial probability
Damper status	*normal*	27/57
	stuck open	10/57
	stuck close	10/57
	stuck in between	10/57
Temperature difference status	*normal*	2/3
	faulty	1/3
Pressure status	*normal*	2/3
	faulty	1/3
Damper position status	*normal*	2/3
	faulty	1/3
Temperature difference	$[0, 1]$	uniform
Pressure	$[0, 1]$	uniform
Damper position	$[0, 1]$	...
Observed temperature difference	[0, 1]	...
Observed pressure	[0, 1]	...
Observed damper position	[0, 1]	...

Table D.1: Cardinality of the variables and their initial values in the first time slice for the damper diagnostic problem shown in Fig. 6.8

Damper status $t(i-1)$	*normal*	*stuck open*	*stuck close*	*stuck in between*
normal	$1-a$	$a/3$	$a/3$	$a/3$
stuck open	$9a/10$	$1-a$	$a/20$	$a/20$
stuck close	$9a/10$	$a/20$	$1-a$	$a/20$
stuck in between	$9a/10$	$a/20$	$a/20$	$1-a$

Table D.2: Conditional probability $p(x_{\mathrm{DS2}}|x_{\mathrm{DS1}})$ of the variable "Damper status $t+1$" depending on the variable "Damper status t". Parameter a was 0.005 in our experiments.

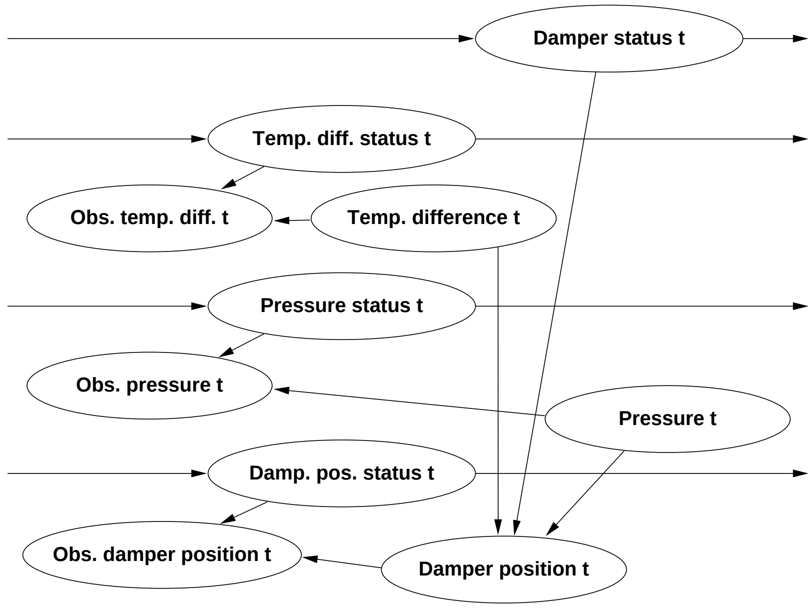


Figure D.1: "Damper" BN also shown in Fig. 6.8.

Status $t(i-1)$	*normal*	*faulty*
normal	$1-b/2$	$b/2$
faulty	b	$1-b$

Table D.3: Dependence of all other status variables besides "Damper status" in two consecutive time slices. Parameter b was 0.005 in our experiments.

Observed variable	*normal*	faulty
x_{OTD}	$N(x_{\mathrm{OTD}}-x_{\mathrm{TD}};0,\sigma^2_{\mathrm{OTD}})$	$c+(1-c)N(x_{\mathrm{OTD}};0,\sigma_0^2)$
x_{OP}	$N(x_{\mathrm{OP}}-x_{\mathrm{P}};0,\sigma^2_{\mathrm{OP}})$	$c+(1-c)N(x_{\mathrm{OP}};0,\sigma_0^2)$
x_{ODP}	$N(x_{\mathrm{ODP}}-x_{\mathrm{DP}};0,\sigma^2_{\mathrm{ODP}})$	$c+(1-c)N(x_{\mathrm{ODP}};0,\sigma_0^2)$

Table D.4: Dependence of observed sensor variables: temperature $p(x_{\mathrm{OTD}}|x_{\mathrm{TD}})$, pressure $p(x_{\mathrm{OP}}|x_{\mathrm{P}})$, and position $p(x_{\mathrm{ODP}}|x_{\mathrm{DP}})$. Parameter c was 0.005 and the corresponding standard deviations σ were 0.1 in our experiments.

where the function $F(x_{\mathrm{TD}}, x_{\mathrm{P}})$ was fit to experimental data and is given by a mixture of threshold tanh() functions:

$$
\begin{aligned}
F(x_{\mathrm{TD}}, x_{\mathrm{P}}) = &+ 0.656195 \times \tanh(-4.83073 + 9.30677 x_{\mathrm{TD}} - 1.282945 x_{\mathrm{P}}) \\
&- 0.357311 \times \tanh(6.27819 - 12.0383 x_{\mathrm{TD}} - 0.0327363 x_{\mathrm{P}}) \\
&- 0.422305 \times \tanh(-3.72830 + 7.38155 x_{\mathrm{TD}} - 0.774815 x_{\mathrm{P}}) \\
&+ 0.501486 \times \tanh(0.107028 - 0.560008 x_{\mathrm{TD}} - 0.0208127 x_{\mathrm{P}}) \\
&- 0.470692 \times \tanh(-4.10404 + 7.03819 x_{\mathrm{TD}} + 0.05501 x_{\mathrm{P}}) \\
&- 0.590046 \times \tanh(1.11134 - 1.44575 x_{\mathrm{TD}} - 0.0181029 x_{\mathrm{P}}) \\
&+ 0.940161,
\end{aligned}
$$

and σ_{DP} was 0.04 in our experiments. For the *"stuck open"* and *"stuck close"* the damper position is $N(x_{\mathrm{DP}}|0, \sigma^2_{\mathrm{DP}})$ and $N(x_{\mathrm{DP}}|1, \sigma^2_{\mathrm{DP}})$ correspondingly. Finally, for the "stuck in between" status variable, the damper position probability distribution was a stepwise distribution with two steps at 0.45 and 0.92. The probability density value was 0.06 for $x_{\mathrm{DP}} < 0.45$ and $x_{\mathrm{DP}} > 0.92$ and $0.94/0.47 = 2$ for $0.45 \leq x_{\mathrm{DP}} \leq 0.92$.

Bibliography

[Alag and Agogino, 1996] Satnam Alag and Alice M. Agogino. Inference using message propagation and topology transformation in vector Gaussian continuous networks. In Eric Horvitz and Finn Jensen, editors, *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 20 – 27. Morgan Kaufmann, 1996.

[Arnborg *et al.*, 1987] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embedding in a k-tree. *Journal of SIAM, Algebraic Discrete Methods*, 8:177 – 184, 1987.

[Bayes, 1763] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans.*, 3:370 – 418, 1763. Reproduced in *Two Papers by Bayes*, W. E. Deming (ed.), New York: Hafner, 1963.

[Boutilier *et al.*, 1996] C. Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In Eric Horvitz and Finn Jensen, editors, *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 115 – 123. Morgan Kaufmann, 1996.

[Cooper, 1990] G. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393 – 405, 1990.

[Cover and Thomas, 1991] T.M. Cover and J.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Chichester, UK, 1991.

[Dagum and Luby, 1993] Paul Dagum and Michael Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60:141 – 153, 1993.

[Dagum and Luby, 1997] P. Dagum and M. Luby. An optimal approximation algorithm for Bayesian inference. *Artificial Intelligence*, 93:1 – 27, 1997.

[Draper and Hanks, 1994] Denise D. Draper and Steve Hanks. Localized partial evaluation of belief networks. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 170 – 177. Morgan Kaufmann, 1994.

[Draper, 1995] Denise L. Draper. Clustering without thinking about triangulation. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 125 – 133. Morgan Kaufmann, 1995.

[Driver and Morrel, 1995] Eric Driver and Darryl Morrel. Implementation of continuous Bayesian networks using sums of weighted Gaussians. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 134 – 140. Morgan Kaufmann, 1995.

[Fung and Del Favero, 1994] Robert Fung and Brendan Del Favero. Backward simulation in Bayesain networks. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 227 – 234. Morgan Kaufmann, 1994.

[Goldschmidt, 1993] Stephen R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, 1993.

[Hastings, 1970] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrica*, 57(1):97 – 109, 1970.

[Heckerman and Breese, 1994] David E. Heckerman and John S. Breese. A new look at causal independence. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 286 – 292. Morgan Kaufmann, 1994.

[Heckerman and Wellman, 1995] David E. Heckerman and Michael P. Wellman. Bayesian networks. *Communications of the ACM*, 38(3):27 – 30, March 1995.

[Heckerman *et al.*, 1992] David E. Heckerman, Eric J. Horvitz, and B.N. Nathwani. Toward normative expert systems. I. the Pathfinder project. *Methods of Information in Medicine*, 31(2):90 – 105, 1992.

[Heckerman, 1990] David E. Heckerman. *Probabilistic Similarity Networks*. PhD thesis, Stanford University, 1990.

[Henrion, 1988] Max Henrion. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In *Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence (UAI–88)*, pages 149 – 163, 1988.

[Henrion, 1991] Max Henrion. Search-based methods to bound diagnostic probabilities in very large belief nets. In B. D'Ambrosio, P. Smeth, and P. Bonissone, editors, *Proceedings of the Seventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–91)*, pages 142 – 150. Morgan Kaufmann, 1991.

[Hoare, 1994] C.A.R. Hoare. Mathematical models for computing science. Available at `http://www.comlab.ox.ac.uk/oucl/users/tony.hoare/publications.html`, August 1994.

[Horvitz and Barry, 1995] Eric Horvitz and Matthew Barry. Display of information for time-critical decision making. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 296 – 305. Morgan Kaufmann, 1995.

[Horvitz and Klein, 1993] E. J. Horvitz and A. C. Klein. Utility-based abstraction and categorization. In David E. Heckerman and Abe Mamdani, editors, *Proceedings* *of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–93)*, pages 128 – 135. Morgan Kaufmann, 1993.

[Huang and Henrion, 1996] Kurt Huang and Max Henrion. Efficient search-based inference for noisy-OR belief networks: TopEpsilon. In Eric Horvitz and Finn Jensen, editors, *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 325 – 331. Morgan Kaufmann, 1996.

[Jaakkola and Jordan, 1996] Tommi S. Jaakkola and Michael I. Jordan. Computing upper and lower bounds on likelihood in intractable networks. In Eric Horvitz and Finn Jensen, editors, *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 340 – 348. Morgan Kaufmann, 1996.

[Jensen *et al.*, 1990] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269 – 282, 1990.

[Kjærulff, 1993] Uffe Kjærulff. *Aspects of Efficiency Improvement in Bayesian Networks*. PhD thesis, Aalborg University, 1993.

[Kjærulff, 1994] Uffe Kjærulff. Reduction of computational complexity in Bayesain networks through removal of weak dependencies. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 374 – 382. Morgan Kaufmann, 1994.

[Kjærulff, 1995] Uffe Kjærulff. HUGS: Combining exact inference and Gibbs sampling in junction trees. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 368 – 369. Morgan Kaufmann, 1995.

[Kozlov and Singh, 1994] Alexander V. Kozlov and Jaswinder Pal Singh. A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Proceedings of the Supercomputing'94 conference*, pages 320 – 329, held in November 1994, Washinton, DC, 1994. IEEE Computer Society Press.

[Kozlov and Singh, 1995] Alexander V. Kozlov and Jaswinder Pal Singh. Sensitivities: an alternative to conditional probabilities for Bayesian belief networks. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 376 – 385. Morgan Kaufmann, 1995.

[Kozlov and Singh, 1996a] Alexander V. Kozlov and Jaswinder Pal Singh. Computational complexity reduction for BN2O networks using similarity of states. In Eric Horvitz and Finn Jensen, editors, *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 357 – 364. Morgan Kaufmann, 1996.

[Kozlov and Singh, 1996b] Alexander V. Kozlov and Jaswinder Pal Singh. Parallel implementations of probabilistic inference. *IEEE Computer*, 29(12):33 – 40, 1996.

[Lauritzen and Spiegelhalter, 1988] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, B 50:253 – 258, 1988.

[Lauritzen and Wermuth, 1989] S. L. Lauritzen and N. Wermuth. Graphical models for association between variables, some of which are qualitative and some quantitative. *The Annals of Statistics*, 17(1):31 –57, 1989.

[Lauritzen, 1992] Stephen L. Lauritzen. Propagation of probabilities, means, and variances in mixed graphical association models. *JASA*, 87(420):1089 – 1108, 1992.

[Lenoski *et al.*, 1990] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148 – 159, May 1990.

[Li and D'Ambrosio, 1994] Zhaoyu Li and Bruce D'Ambrosio. Efficient inference in Bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11(1):55 – 81, 1994.

[Neapolitan, 1990] Richard E Neapolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms.* John Wiley & Sons, New York, 1990.

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

[Poole, 1993] David Poole. The use of conflicts in searching Bayesian networks. In David E. Heckerman and Abe Mamdani, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–93)*, pages 359 – 367. Morgan Kaufmann, 1993.

[Pradhan *et al.*, 1994] Malcolm Pradhan, Gregory Provan, Blackford Middleton, and Max Henrion. Knowledge engineering for large belief networks. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 484 – 490. Morgan Kaufmann, 1994.

[Rothberg, 1993] Edward Eric Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization.* PhD thesis, Stanford University, 1993.

[Rubinstein, 1981] Reuven Y. Rubinstein. *Simulation and the Monte Carlo Method.* John Wiley & Sons, New York, 1981.

[Samet and Webber, 1988] H. Samet and R.E. Webber. Hierarchical data structures and algorithms for computer graphics. I. Fundamentals. *IEEE Computer Graphics and Applications*, 8(3):48 – 68, 1988.

[Shachter *et al.*, 1994] Ross D. Shachter, Stig K. Anderson, and Peter Szolovits. Global conditioning for probabilistic inference in belief networks. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 514 – 522. Morgan Kaufmann, 1994.

[Shachter, 1986] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871 – 882, 1986.

[Shachter, 1990] Ross D. Shachter. Evidential reasoning using likelihood weighting. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence (UAI–90)*, 1990.

[Shoikhet and Geiger, 1997] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In *Proceedings of the AAAI-97 Conference*, pages 185 – 190. Morgan Kaufmann, 1997.

[Shwe *et al.*, 1991] M. A. Shwe, B. Middleton, D. E. Heckerman, M. Henrion, E. J. Horvitz, H. P. Lehmann, and G. F. Cooper. Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base. I. The probabilistic model and inference algorithms. *Methods of Information in Medicine*, 30(4):241 – 255, October 1991.

[Singh, 1993] Jaswinder Pal Singh. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*. PhD thesis, Stanford University, 1993.

[Stenström and Dahlgren, 1996] Per Stenström and Frederic Dahlgren. Applications for shared memory multiprocessors. *IEEE Computer*, 29(12):29 – 31, 1996.

[Wellman and Liu, 1994] Michael P. Wellman and Chao-Lin Liu. State-space abstraction for anytime evaluation of probabilistic networks. In Ramon Lopez de Montara and David Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–94)*, pages 567 – 574. Morgan Kaufmann, 1994.

[Woo *et al.*, 1995] Steve Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24 – 36, 1995.