AN ABSTRACT OF THE THESIS OF

Zhaoyu Li for the degree of Doctor of Philosophy in Computer Science

presented on June 8, 1993.

Title: A Factoring Approach For Probabilistic Inference In Belief Networks

Abstract approved: Redacted for Privacy

Bruce D'Ambrosio.

Reasoning about any realistic domain always involves a degree of uncertainty. Probabilistic inference in belief networks is one effective way of reasoning under uncertainty. Efficiency is critical in applying this technique, and many researchers have been working on this topic. This thesis is the report of our research in this area.

This thesis contributes a new framework for probabilistic inference in belief networks. The previously developed algorithms depend on the topological structure of a belief network to perform inference efficiently. Those algorithms are constrained by the way they use topological information and may not work efficiently for some inference tasks. This thesis explores the essence of probabilistic inference, analyzes previously developed algorithms, and presents a factoring approach for probabilistic inference. It proposes that efficient probabilistic inference in belief networks can be considered as an optimal factoring problem.

The optimal factoring framework provides an alternative perspective on probabilistic inference and a quantitative measure of efficiency for an algorithm. Using this framework, this thesis presents an optimal factoring algorithm for poly-tree networks and for arbitrary belief networks (albeit with exponential worst-case time complexity for non-poly-tree networks). Since the optimal factoring problem in general is a hard problem, a heuristic algorithm, called set-factoring, is developed for multiply-connected belief networks. Set factoring is shown to outperform previously developed algorithms. We also apply the optimal factoring framework

to the problem of finding an instantiation of all nodes of a belief network which has the largest probability and present an efficient algorithm for that task.

Extensive computation of probabilistic inference renders any currently used exact probabilistic inference algorithm intractable for large belief networks. One way to extend this boundary is to consider parallel hardware. This thesis also explores the issue of parallelizing probabilistic inference in belief networks. The feasibility of parallelizing probabilistic inference is demonstrated analytically and experimentally. Exponential-time numerical computation can be reduced by a polynomial-time factoring heuristic. This thesis offers an insight into the effect of the structure of a belief network on speedup and efficiency.

A Factoring Approach For Probabilistic Inference In Belief Networks

by

Zhaoyu Li

A Thesis submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Completed June 8, 1993
Commencement June 1994.

Approved:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented June 8, 1993

Typed by Zhaoyu Li

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my major advisor Professor Bruce D'Ambrosio for all the help and support he provided me throughout this research. I would also like to express my appreciation to my other committee members, Professors Thomas Dietterich, Vikram Saletore and Prasad Tadepalli, for their assistance.

I would like to thank several friends and fellow graduate students for their assistance. Thanks to the family of Ackerson's – Gale, Duane and Robert – for their help with the English of my papers. Thanks to the other graduate students of Professor D'Ambrosio – Tony Foutain, Daphne Yuan and Caryl Westerberg – for all kinds of help.

I would like to express my gratitude to my parents, parents-in-law and brothers, for being endless source of love and support to me. The last, and surely the most important, note of thanks belongs to my dearest wife – Wanhong. Without her patience, understanding and personal sacrifice, none of this would have been possible.

Table of Contents

Table of Contents (Cont.)

Table of Contents (Cont.)

List of Figures

List of Tables

A Factoring Approach For Probabilistic Inference In Belief Networks

Chapter 1

Introduction

1.1 Motivation

Over the last few years, a method of reasoning using belief networks has become popular within the AI probability and uncertainty community. This method provides a formalism for reasoning about beliefs under conditions of uncertainty. In this formalism, propositions are given numerical parameters signifying the degree of belief and the parameters are combined and manipulated according to the rules of probability theory. Belief networks have been applied to problems in medical diagnosis [HBH89, SFB89], map learning [Dea90], language understanding [CG89b, CG89a, Gol90], vision [LAB89], heuristic search [HM89], and so on.

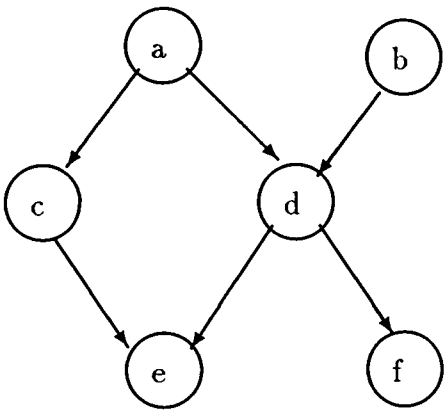
A number of exact algorithms have been developed to perform probabilistic inference in belief networks in recent years. These algorithms depend on the topological structures of belief networks. Among them are the propagation algorithms based on the original directed graph, such as the poly-tree propagation algorithm [Pea88, Pea86, Peo91], or on a related directed graph, such as the conditioning algorithm [Pea88] and the symbolic probabilistic inference algorithm [D'A89, SDD90],

or on a related undirected graph, such as the clustering algorithm [Pea88] and algorithms in [LS88] and [JOA90], and the reduction algorithms [Sha86, Sha88, Sha89].

It is desirable to compare these algorithms and analyze the advantages and disadvantages of them and to find more efficient algorithms. The motivation of our research is to explore the essence of different algorithms and to design and implement more efficient ones for probabilistic inference in belief networks. There are two steps in our research. The first step is to analyze typical algorithms of different kinds of methods for probabilistic inference and to summarize characteristics relevant to the efficiency of these algorithms. The second step is to explore a new approach and to design new algorithms. Probabilistic inference may involve different kinds of belief networks, standard belief networks and non-standard belief networks; involve different inference tasks, such as querying the marginal probability of a variable and the special task of querying the most probable explanations of a belief network; and involve different computational models, such as sequential probabilistic inference and parallel probabilistic inference. The task of our research is to consider the influence of these factors and to solve different sub-problems.

1.2 Background

Belief networks provide an intuitively appealing knowledge representation for probabilistic models. A belief network is a directed acyclic graph containing a set of nodes, a set of arcs, and a set of numeric probability distributions. A node represents a domain variable (or a proposition) with mutually exclusive and collectively exhaustive values. Arcs, and associated numeric probability distributions, describe probabilistic relationships between the nodes. The arcs signify the existence of direct causal influences between the linked variables and the strengths of these influences are quantified by conditional probabilities. A singly-connected belief network is a belief network in which there is no more than one undirected path between any two nodes. A multiply-connected belief network is a belief network in which there



p(a): p(a=1)=0.2
p(b): p(b=1)=0.3
p(c|a): p(c=1|a=1)=0.8 p(c=1|a=0)=0.3
p(d|a,b): p(d=1|a=1,b=1)=0.7
p(d=1|a=1,b=0)=0.5
p(d=1|a=0,b=1)=0.5
p(d=1|a=0,b=0)=0.2
p(e|c,d): p(e=1|c=1,d=1)=0.5
p(e=1|c=1,d=0)=0.8
p(e=1|c=0,d=1)=0.6
p(e=1|c=0,d=0)=0.3
p(f|d): p(f=1|d=1)=0.2 p(f=1|d=0)=0.7

Figure 1. A simple multiply-connected belief network.

are at least two nodes with more than one undirected path. Figure 1 is a simple multiply-connected belief network. [1]

Probabilistic inference in a belief network refers to the process of querying the probability of a set of variables in the belief network, given evidence on some subset of the remaining variables. The most common questions we ask in a belief network are: marginal probability of a node x, conditional probability of x given y, and joint probability of a set of variables in a belief network. In the simple belief network in Figure 1, for example, we may query the marginal probability of variable e, i.e., $p(e)$; query the conditional probability $p(e|b = 1)$; and query the joint probability $p(d = 1, e = 0, f = 1)$. We may also query the marginal probability of each node or the most probable explanation of a belief network, and so on.

The result of evaluating a belief network can be exact or approximate. There are some reasons for considering exact methods in probabilistic inference in belief networks. First, approximate solutions may not be as satisfactory as exact solutions for some problems. Second, belief networks which model realistic problems or systems may not have high connectivity; so efficient exact algorithms may be

[1] In this thesis, we will interchangeably use the term node and variable.

tractable. Third, parallel computing architectures assure that some computations are acceptable even if they are not acceptable in a sequential computing architecture.

The reason for choosing an approximate solution instead of an exact solution for a query in probabilistic inference is the intractability of exact solutions for very large belief networks with high connectivity. There are many ways to find approximate solutions to a query in a belief network. These approximate algorithms have a lot in common. Essentially, they randomly posit values for some of the variables in a network and then use them to pick values for the other variables. Statistics on the values of these variables give the answer to a query. In this paper, we consider exact solutions, exclusively, for all queries.

A belief network represents a full joint probability distributor over the n domain variables in the belief network. In particular, the full joint probability distribution can be calculated as follows [Pea88, Sha86]:

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

Where x_1,..., x_n are n variables in the belief network; π_i is the set of direct predecessors of x_i; $p(x_i|\pi_i)$ is the conditional probability for variable x_i if π_i is not the empty set, otherwise it is the marginal probability of x_i. The product of any two terms of the formula is usually called *a conformal product*, the number of variables appearing in a conformal product is called *the dimension* of the conformal product, and the maximum number of variables in any of the conformal products for a query is called *the maximum dimensionality* of the conformal products (or the query), or the "dimensionality" for short. The time complexity of computing the full joint probability distribution of a belief network is exponential in the number of nodes of the network.

An important characteristic of a belief network is that any conditional, marginal, or conjunctive query in the belief network can be calculated from the full joint probability by using Bayes Theorem. This can be done by instantiat-

ing observed variables in the formula and summing over the variables that are not queried after the conformal products. It should be noted that the computation time can be reduced if a variable can be summed over when it appears only in one distribution (if these variables are not queried), instead of carrying out the operation in the last step of computation. For example, we want to query the marginal probability $p(d)$. Computing $p(d) = \sum_a(\sum_b(p(d|a,b)p(b))p(a))$ needs less number of multiplications than that for computing $p(d) = \sum_{a,b}((p(d|a,b)p(b))p(a))$. Therefore, the efficiency of probabilistic inference in a belief network depends on the ordering of distributions of related variables.

The influence of an observation on a belief net can be reflected by instantiating the observed node in the distributions that contain it and by removing the arcs connecting the observed node with its child nodes. A query after observations in a belief network can be considered as a query directly on the modified belief network with respect to the observations. For simplicity, therefore, we may talk about an evidence or an observation in a belief network, but we will not consider any observations in our discussion in the rest of the paper. The algorithms designed for non-observations are the same as those for observations. For the sake of discussion, we assume that all variables have domain size two; any conclusion based on this assumption applies to the case of unequal domain size variables, unless we indicate otherwise.

Some variables may not be relevant to every query in a belief network. In this case, we can save some computation time if we just consider the variables relevant to the query instead of computing the conformal products for the full joint probability distribution. Theoretical research in [GVP89, Pea88] provides a way of finding relevant variables to a query in a belief network in linear time in the total number of variables. The variables related to the query correspond to a new belief network in which the query can be obtained by computing the full joint probability of the new belief network first and then summing over the non-queried variables. In this paper, we assume that all nodes we consider are relevant to the query, since we

can find relevant variables to a query in linear time with respect to the number of nodes in a belief network [SDD90]. That is, we ignore the linear time computation and focus our attention on the exponential probability computation.

The effectiveness and efficiency of probabilistic inference in a belief network have been the subject of considerable research in recent years. The time complexity of probabilistic inference in belief networks can be viewed as having two components.

First, it has been proved that probabilistic inference in belief networks is an NP-hard problem [Coo90a]. That is, it is unlikely that a general, efficient probabilistic inference algorithm can be developed for an arbitrary belief network; however, some special algorithms for particular structures of belief networks can be developed. For example, the poly-tree propagation algorithm for a singly-connected belief network [Pea88] requires only polynomial time. If we consider structures of belief networks more carefully, we will find that the hardness of the problem depends on the structure of a network. It is easy to know that a query in a singly-connected belief network and a fully connected belief network can be handled in polynomial time with respect to the input size of the problem. This indicates that the hardness of probabilistic inference in belief networks lies in those belief networks neither fully connected nor singly-connected. Unfortunately, most practical belief networks are neither fully connected nor singly-connected. However, they are very sparse.

Second, the computational cost of probabilistic inference in a belief network is exponential in the number of the variables in the graph with respect to the worst case. This happens to almost fully connected belief networks. We do not expect to find any algorithm which is much more efficient than any other algorithm for these kinds of belief networks. Fortunately, practical belief networks are rarely fully connected and we can take advantage of the conditional independence represented by sparseness of the graphs to reduce the computation cost. Since multiplication is a basic operation in probabilistic inference and the other operations are much less frequent and with less cost (compared with the cost of multiplication)[2], we will

[2]This statement is true with standard belief networks. If considering noisy-or networks, the

hereafter consider the number of multiplications as a measure of the computational cost of the probabilistic inference in a belief network.

1.3 Overview of the thesis

The remainder of the thesis is organized as follows. Chapter two is a review of some previously developed exact probabilistic inference algorithms. Not all existing algorithms will be covered in this chapter, but examples of all basic methods for probabilistic inference are described in the review. The review of these algorithms will show briefly the ideas of different exact algorithms and summarize the characteristics of the algorithms and the advantages and the disadvantages of using those algorithms.

Chapter three presents an optimization problem, optimal factoring, which is the basis of our new framework for finding efficient algorithms for probabilistic inference in belief networks. We will give a formal definition of the optimal factoring problem and discuss the characteristics of the problem. Since the optimal factoring problem is hard, we do not expect to find an efficient algorithm to solve it in general. Instead, we can find different factoring strategies for different instances of the problem. We will present some factoring strategies for some simple instances of the problem, and present an optimal factoring strategy for an arbitrary instance but with exponential computational cost with respect to the number of variables considered. Finally, we will discuss the mapping between the optimal factoring problem and probabilistic inference. The purpose of introducing the optimal factoring problem is to apply the techniques used for solving optimization problems to the optimal factoring problem and to apply the results of the factoring problem to probabilistic inference; so that we can get more efficient algorithms for probabilistic inference. From the factoring point of view, previously developed algorithms are just

number of additions in the computation can not be ignored.

different factoring strategies. However, these factoring strategies are constrained by graphical structures and may not be efficient in some cases.

Chapter four discusses the issue of applying the results of the optimal factoring problem to probabilistic inference. First, we will present an optimal factoring algorithm for singly-connected belief networks based on the factoring strategies proposed in chapter 3. Second, we will discuss factoring for multiply-connected belief networks. Factoring for multiply-connected belief networks is a hard problem. We will discuss some factoring strategies for multiply-connected belief networks and turn our attention to finding heuristic factoring algorithms. Third, we will present a heuristic factoring algorithm for multiply-connected belief networks. And finally, we will show, by experimental tests, that a simple, greedy algorithm outperforms previously developed algorithms.

In chapter five, we will explore another problem relevant to belief networks: finding the l most probable explanations of a belief network given a set of evidence. Given a belief network with evidence, the task of finding the l most probable explanations (MPE) in the belief network is that of identifying and ordering the l most probable instantiations of the non-evidence nodes of the belief network. Although many approaches have been proposed for solving this problem, most work only for restricted topologies (i.e., singly-connected belief networks). In this chapter we will present a framework, optimal factoring, for finding the l MPEs in arbitrary belief networks. Under this framework, efficiently finding the MPE in a belief network can be considered as the problem of finding an ordering of distributions of the belief network and efficiently combining them. We will discuss the essence of the problem of finding the MPEs and present an optimal algorithm for singly-connected belief networks and an efficient algorithm for multiply-connected belief networks. We will also discuss the problem of finding the MPE in a set of variables of a belief network under this framework.

Because of the computational complexity of probabilistic inference in belief networks, we turn our attention to parallel hardware for probabilistic inference in

large belief networks. In chapter six, we will discuss the issue of parallelizing probabilistic inference in belief networks using distributed memory architecture. We will show that the problem of parallelizing probabilistic inference is also an optimal factoring problem. From this point of view, we will analyze the features of the problem and present a heuristic factoring algorithm for parallel probability computation. We will present experimental results for hypercube architectures which verify our analysis from the optimal factoring perspective and provide insight into the effect of the maximum dimensionality of a query on speedup and efficiency.

In chapter seven, we will conclude our research with the statement that any task of probabilistic inference in belief networks should be considered as an optimal factoring problem. The discussion in previous chapters supports the statement from the point of view of different tasks in belief networks and different belief networks. In this chapter we will discuss some characteristics of the factoring method in general and show that the factoring method is more general than any other previously developed method for probabilistic inference and provides a useful framework for finding efficient probabilistic inference algorithms.

In chapter eight, we will conclude our research and propose future research in finding efficient algorithms for exact probabilistic inference in belief networks.

Chapter 2

Review of Exact Probabilistic Inference Algorithms

A number of algorithms have been developed to perform probabilistic inference in belief networks in recent years. These algorithms apply different methods to answer marginal, conditional and conjunctive queries[3]. In this chapter, we will briefly review the methods from some typical algorithms and summarize some factors relevant to efficiency of probabilistic inference methods.

2.1 The poly-tree propagation algorithm

The poly-tree propagation algorithm has been developed for probabilistic inference in singly-connected belief networks [Pea88, Peo91]. In a singly-connected belief network, probabilistic information of any node can be propagated from the node to its neighbors and from its neighbors to their neighbors and so on, through the structure of the belief network. If a node receives this kind of information from its parents and its children, its marginal probability can be easily calculated. So, the main step in the poly-tree propagation algorithm is to propagate this kind of information from any node to the rest of the belief network. If we let e_x^- stand for

[3]We use a method to denote a general approach for probabilistic inference in belief networks, an algorithm to denote an implementation of a method. So, there could be many different algorithms in one method.

the evidence contained in the tree rooted at x, e_x^+ stand for the evidence contained in the rest of the network, then x separates e_x^- and e_x^+. Let $\lambda(x)$ and $\pi(x)$ denote $p(e_x^-|x)$ and $p(x|e_x^+)$. Belief distribution of a variable x can be calculated by:

$$BEL(x) = p(x|e_x^+, e_x^-) = \alpha p(e_x^-|x, e_x^+)p(x|e_x^+) = \alpha p(e_x^-|x)p(x|e_x^+) = \alpha\lambda(x)\pi(x),$$

here $\alpha = [p(e_x^-|e_x^+)]^{-1}$ is a normalizing constant rendering $\sum_x BEL(x) = 1$ [Pea88].

There are three steps in the poly-tree propagation algorithm for local propagation and belief computation, which can be executed in any order. Assuming that a typical node x has n parents u_1,..., u_n and m children y_1, ...y_m, the belief distribution of variable x can be computed provided that three types of distributions for node x are available:

1. The current π message contributed by each parent u_i:

$$\pi_x(u_i) = p(u_i|e_{u_i x}^+)$$

2. The current lambda message contributed by each child y_j:

$$\lambda_{y_j}(x) = p(e_{xy_j}^-|x)$$

3. The conditional probability distribution $p(x|u_1, ..., u_n)$.

where $e_{u_i x}^+$ stands for the observations contained in the ancestors of nodes u_i, and $e_{xy_j}^-$ stands for the observations contained in the sub-tree rooted at node y_j. Then three steps can be performed as:

1. Belief updating: the belief distribution of variable x, $BEL(x)$, is $BEL(x) = \alpha\lambda(x)\pi(x)$ where

$$\lambda(x) = \prod_j \lambda_{y_j}(x),$$

$$\pi(x) = \sum_{u_1,...,u_n} p(x|u_1, ..., u_n) \prod_i \pi_x(u_i).$$

2. Bottom-up propagation: node x computes a new λ message to be sent to its parents u_i:

$$\lambda_x(u_i) = \beta \sum_x \lambda(x) \sum_{u_k, k \neq i} p(x|u_1, ..., u_n) \prod_{k \neq i} \pi_x(u_k)$$

3. Top-down propagation: node x computes the new π message to be sent to its child y_j:

$$\pi_{y_j}(x) = \alpha(\prod_{k \neq j} \lambda_{y_k}(x)) \sum_{u_1, ... u_n} p(x|u_1, ..., u_n) \prod_i \pi_x(u_i)$$
$$= \alpha BEL(x)/\lambda_{y_j}(x).$$

The algorithm can be executed in parallel, or sequentially executed in any order for each node if the π and λ messages for it are ready. When a belief network is initialized, the π message of each root node is equal to its prior probability $p(x)$ and all childless nodes are assigned the lambda message as $\lambda(x) = (1, \ldots, 1)$. Then the algorithm starts λ and π propagations and belief computations. When observations are inserted, all observed nodes are set as the lambda message as (0, ...,0,1,0, ...,0) with 1 at the position that the observation is true, and then the propagation is activated. Given a new observation, only π propagation is needed for the sub-tree rooted at the observed node, and λ propagation for its antecedents followed by π propagation for the other children of the antecedents.

From the three steps above, based on the domain size of each variable and on the number of parents of each node, we see that the time complexity of λ, π and belief distribution computations for each node is exponential in the number of parent variables, because the summation in each formula ranges over all value combinations of parent variables. The computation time of the algorithm is affected by the location of observations, since the positions of observations may reduce the number of parents of a node, and different observations result in different propagations.

We observe the following characteristics of the algorithm. First, since the propagations in the algorithm are carried out through the structure of a belief network, namely through the neighbors of a node, conformal products are performed

only between nodes directly connected by arcs. There are no propagations among siblings. Second, the algorithm doesn't prescribe the order for λ and π propagations. Third, the propagations go through whole networks. And finally, the algorithm is incremental with respect to the observations.

The limitation of the algorithm is that it can only be applied to a singly-connected belief network. This algorithm is not applicable to a multiply-connected belief network because, for any such network, there exists at least one pair of nodes with more than one pathway between them. If the propagation algorithm were applied to the multiply connected network, one node would receive a message more than once from the other node, and the computation of marginal probability would not be correct.

2.2 Conditioning algorithm

Since most belief networks in practical use are not singly connected, several methods for converting a multiply-connected belief network into a singly-connected belief network have been developed. One of the methods is called conditioning [Pea88]. It is based on the idea of changing the connectivity of a belief network and rendering it singly-connected by instantiating a selected group of variables.

An intuitive way of converting a multiply-connected directed graph to a singly-connected graph is to remove arcs in the graph such that no more than one directed pathway exists between any two nodes. The process can be performed in a belief network by instantiating a variable in a pathway instead of removing an arc, since an instantiated variable in a pathway can block information passing between its parents and its children and among its children. The set of variables to be instantiated is called a *cutset*. Given a multiply connected belief network with variables x_1, x_2, ..., x_n, the method of conditioning for querying a variable x given observations E is as follows. First, choose a group of variables x_{i_1}, x_{i_2},..., x_{i_k} from the network, which, if removed from the graph, render the remaining subgraph

singly-connected. Second, instantiate them and apply the poly-tree propagation algorithm. Obviously, the number of instantiations of the chosen variables is the product of the domain size of each variable; that is, the propagation algorithm is invoked exponentially many times (in terms of the number of variables to be instantiated). The final result of inference is determined by summing the intermediate results weighted by the posterior probabilities of the instantiations. The formula for calculating the query is:

$$p(x|E) = \sum p(x|E, x_{i_1}, ..., x_{i_k}) p(x_{i_1}, ..., x_{i_k}|E).$$

in which, $p(x|E, x_{i_1}, ..., x_{i_k})$ and $p(x_{i_1}, ..., x_{i_k}|E)$ can both be calculated by the poly-tree algorithm.

Since the computation time is exponential in the number of variables in a cutset, a minimal cutset of a belief network is desirable. Unfortunately, finding a minimal cutset in a belief network is NP-hard[SC88]. A heuristic algorithm has been proposed for finding a cutset in polynomial time with respect to the number of nodes[SC88]. The main steps of the algorithm are the following:

1. Remove all nodes that have a single parent and no children and the nodes that have a single child and no parent; remove the arcs that connected the pruned nodes. Repeat the step until it can no longer be applied.

2. If there are any nodes left, find a good cutset candidate. Three steps for choosing a good cutset candidate are:

   (a) Choose the nodes that have one or no parents;

   (b) Choose the node from (a) that has the most neighbors;

   (c) If there is more than one node, choose the node that has the lowest number of possible values.

   Add the node chosen above into the cutset, then remove it from the network. If there remain nodes in the network, return to the beginning.

This algorithm does not guarantee finding a minimal cutset. An experiment with 60 randomly generated belief networks showed that it found the minimal cutset in approximately 70 percent of the networks[SC88]. The conditioning algorithm is exponential in cutset size, which determines the number of times to apply the poly-tree algorithm, and is exponential in the maximum number of antecedents of a node in the remaining poly-tree when using the poly-tree algorithm for each instantiated network.

The following are the characteristics of the conditioning algorithm. First, the algorithm is based on the poly-tree algorithm; therefore, it inherits characteristics of the poly-tree algorithm. Second, performance of the algorithm depends on the performance of the heuristic algorithm for finding a cutset; that is, the computational complexity of the algorithm is exponential with respect to the number of variables in a cutset. Since finding a minimal cutset in a belief network is an NP-hard problem, we don't expect a heuristic algorithm to guarantee a minimal cutset for every query in any belief network. Finally, the conditioning algorithm is not incremental with respect to observations because the cutset should be different with new observations inserted.

2.3 Clustering algorithm

Another method of converting a multiply-connected belief network to a singly-connected belief network is clustering. Clustering involves combining several variables in a multiply-connected network to form a compound variable such that a new network of compound variables, as nodes, is singly-connected, and then applying the poly-tree propagation algorithm. Generally, there are many different ways to cluster a belief network since any two nodes in a belief network can be merged into one node if there is at most one directed path connecting the two nodes. In the extreme case, a clustered tree can be formed by lumping together all non-leaf variables as one variable. However, the exponential cost in the number of variables to be

combined to form a compound variable and the structureless nature of a converted belief network make it difficult to compute and explain the beliefs accrued by this variable[Pea88]. There are many papers about different clustering strategies. We describe one strategy discussed in [Pea88] here.

There are two parts in the clustering algorithm. First, create cliques (a clique is a maximal subset of nodes which is a complete sub-graph) and form a join tree (a tree with cliques as nodes). Second, update the tree when any observations are inserted. Creating cliques and forming a join tree involve the following steps:

1. Convert the belief network into a Markov network by interconnecting the parents of each node and then making the original directed edges in the belief network non-directed. A Markov network is an undirected graph in which directly unconnected nodes are conditionally independent.

2. Triangulate the network by using the maximum cardinality search, which transforms the belief network into a chordal graph (a chordal graph is a undirected graph in which every cycle of length of four or more has a chord). After triangulation, the chordal graph can be decomposed into a set of cliques that have the running intersection property. The running intersection property here means that when all cliques are sorted in an order of $(C_1, C_2, ..., C_n)$, then for all $j \geq 2$, there exist $i < j$ such that $C_i \subseteq C_j \cap (C_1 \cup ... \cup C_{j-1})$.

3. Sort the cliques in the increasing order of the maximum index number of nodes[4] in a clique and assemble them in a join tree by connecting each clique node C_i to a clique node C_j $(j < i)$ sharing the maximum index number of nodes with C_i . Each clique is now modeled as a clique node in the join tree.

4. Set up the conditional probability distributions of clique nodes according to the links in the join tree. Each combination of states of

[4]Every node is indexed by the maximum cardinality search algorithm[TY84] in the step of creating a Markov network.

the component belief network nodes of the clique node is one state of the clique node.

The time complexity of step 1 to step 3 is polynomial in the number of nodes in the belief network. Step 4, setting up probability distributions, is time consuming. The states of a join node are determined by the combination of all belief nodes in that join node, so the operations required to compute the probability distributions for a join node are exponential in clique size.

Once a join tree is formed, the poly-tree algorithm can be applied. Note that a belief network node can be in more than one clique. An observation on that node may affect more than one clique node, thus the number of steps in propagating an observation in a clique tree may be greater than in a singly connected belief network. When the probability distribution is obtained for each clique node, the probability for original belief network nodes can be calculated easily. The calculation for the beliefs of each of the belief network nodes is done by finding the smallest clique node of which the belief network node is a member, and then marginalizing the clique node's belief over the states of the other belief nodes of the clique node.

The characteristics of the clustering algorithm are as follows. First, the algorithm is based on the poly-tree algorithm; therefore, it inherits some characteristics from the poly-tree algorithm. Second, performance of the algorithm depends on the structure of the clique tree; that is, the computational complexity of the clustering algorithm is exponential with respect to the maximum number of variables in a clique. Finally, the conditioning algorithm is incremental with respect to observations, since new observations can be handled by modifying the clique nodes containing the observed nodes.

2.4 Reduction algorithm

There are several algorithms that directly handle probabilistic inference in a multiply-connected belief network. One of them is called the reduction algorithm [Sha86,

Sha88]. The main idea of the reduction algorithm is the following: given a belief network with n nodes representing probabilistic relations of n variables x_1, x_2, ..., x_n, the marginal probability of any variable x_i given observations x_{i_1}, ..., x_{i_k}, that is $p(x_i|x_{i_1}, ..., x_{i_k})$, corresponds to a particular graph, and this graph can be derived from the original belief network by node removal and arc reversal. These transformations maintain the consistency between the original network and the transformed network for the query.

A heuristic algorithm has been developed for the reduction process[RA88]. The algorithm requires $O(n^2)$ storage and $O(n^3)$ time for symbolic or topological transformation. The symbolic transformation algorithm removes nodes and reverses arcs until only conditioning and conditional nodes are left. Assuming that C is the set of conditioning and conditional nodes and K is the other nodes to be removed, the process of the reduction algorithm is:

1. Remove all nodes in K that have no successors. Repeat until all nodes without successors are removed. If K is empty go to 5.

2. Remove all nodes in K that have only one successor each. If more than one node is to be removed, remove them in the order of fewer predecessor first. Repeat until all nodes with one successor are removed. If K is empty go to 5.

3. Select the node in K with the least number of successors, let the node be x.

4. Pick a successor of node x and check the number of paths from x to the successors; if there is more than one path, reject it; otherwise reverse the arc between x and that successor with concomitant updating of their predecessor and successor set according to Bayes theorem. Check if x has only one successor, if so remove x and if K is not empty go to 2 otherwise go to 5. If x has more than one successor go to 4.

5. For every successor of a conditional node, check for more than one path between conditional node and that successor. If there is more than one path,

reject that successor; if not, keep track of the successor with the least number of predecessors y. Reverse the arc between the conditional node and y with a concomitant updating of successor and predecessor sets. Repeat 5 until the conditional node has no successors.

Most of the computational cost in the algorithm comes from calculating the new conditional probability distributions in arc reversal. Assume that node x_i conditionally depends on nodes w_i, node x_j conditionally depends on node x_i and nodes w_j, and both x_i and x_j conditionally depend on nodes w_{ij}, then after arc reversal between node x_i and x_j, the conditional probability of $P(x_j|w_i, w_j, w_{ij})$ and $P(x_i|w_i, w_{ij})$ will be [Sha86]:

$$p(x_j|w_i, w_j, w_{ij}) = \sum_{x_i} p(x_j|x_i, w_j, w_{ij})p(x_i|w_i, w_{ij})$$

$$p(x_i|x_j, w_i, w_j, w_{ij}) = p(x_j|x_i, w_j, w_{ij})p(x_i|w_i, w_{ij})p(x_j|w_i, w_j, w_{ij}).$$

From the above formulas we know that the cardinalities of x_j and x_i are usually increased after arc reversal and that calculation of the conditional probability distribution is exponential in the number of parents of node x_j and x_i. We use the term *dimensionality* of node x_i to denote the number of arcs from the set of nodes x_j, w_i, w_j, w_{ij} to node x_i. The probability computation for node x_i will be exponential in its dimensionality.

The characteristics of the reduction algorithm can be summarized as follows. First, the computational complexity of the algorithm is exponential with respect to the maximum dimensionality, which is determined by a heuristic strategy, and may not be the same as the complexity of the query by using some other algorithms. Second, the algorithm is not incremental with respect to the observations. Third, caching is difficult to implement in the algorithm and any query will start transformation from the original belief network.

2.5 The Symbolic Probabilistic Inference (SPI) algorithm

The symbolic probabilistic inference algorithm (SPI) [D'A89, SDD90] is a query-driven algorithm which uses Bayes theorem directly for probabilistic inference in multiply-connected networks. In this algorithm, probabilistic inference occurs in three steps. In the first step, a partition tree[5] is created from the belief network. In the second step, symbolic reasoning based on the partition tree determines which nodes to combine and how these nodes should be combined, given observed nodes and queried nodes. In the last step, the probability computations are carried out for those nodes according to the structure of the partition tree. The second and third steps may mix together within a partition.

For symbolic reasoning, a node in SPI is represented in terms of its conditional distribution and the marginal distributions of its immediate antecedents. When a node is queried, its marginal probability can be calculated from its symbolic expression if all marginal probabilities of its immediate antecedents' are known (the joint marginal probability for the antecedents in the same subtree is computed for computing the queried node). If some of them are unknown, then sub-queries are generated for these nodes. The partition tree provides a structure for generating sub-queries and efficient probability computation. Associated with each partition node in the partition tree is a *conditioning* expression which stores the union of the expressions for the observed value of each observed variable in that partition. There is one conditioning expression for each partition node because any observation in a partition will only affect its child partition node. In order to speed up probability computation, an ordering heuristic algorithm is used in each partition before probability computation. The partition tree also provides a way of caching intermediate computation results. In the case of multiple queries, when a query proceeds in the same path as a previous query, the previously cached results can

[5] A partition tree is a tree in which a node is a subset of nodes of the original belief network. A partition strategy determines the subset of nodes.

be used. The cache invalidation strategy used in the algorithm is to invalidate all caches on the path from that partition to the root.

The computational complexity of the SPI algorithm is mainly determined by the third step. Step one is a heuristic partitioning algorithm; its complexity is $O(n^2)$. In step two, after each observation, the algorithm will change the representations of the children of the observed node. The algorithm resets the expression of the observed node such that its value space is a singleton consisting of only the observed value. It also resets the conditioning expression to the union of its current expression and the expression for the observed value of the observed node. Hence, the representation in SPI is incremental in observations. The symbolic reasoning in this step is polynomial in the number of partitions in the partition tree. Step 3 is the costliest step, because the time for computing any two expressions is exponential in the number of variables in the two expressions or the dimensionality of the conformal product.

The characteristics of the SPI algorithm can be summarized as follows. First, the computational complexity of the algorithm is exponential with respect to the maximum dimensionality, which is determined by the partition tree and the ordering heuristic strategy. Second, the algorithm is incremental with respect to observations. And third, it has a caching strategy for reducing some repeated computations.

2.6 Summary

Analyzing these exact probabilistic inference algorithms, we can summarize some features that appear in some or all of currently developed exact probabilistic inference algorithms. We focus our attention on those features relevant to the efficiency of these algorithms.

These algorithms are closely related to the structure of belief networks. More precisely, these algorithms rely either on the original directed graph, on a related directed graph, or on a related undirected graph. For example, the poly-tree propagation algorithm relies on the original poly-tree; the rest of the algorithms rely on a related graph. The algorithm developed in [LS88] relies on the related undirected graph. The structure of a graph apparently describes the local and global information of the nodes in the graph. However, verifying some global information through the structure of a graph is usually a hard problem, for example, cutset determination. The structure of a graph also constrains the way nodes are combined or expressions are factored. These algorithms rely on the structure of a graph, but either consider only local information or turn to heuristic methods. This feature may make these algorithms not as efficient as an algorithm that considers global information, not available locally in the structure of the graph.

Numeric computation vs. non-numeric computation. If we refer to the computation of conformal products as *numeric computation* and the rest of each algorithm as *non-numeric computation* or symbolic reasoning, we find that the numeric computation in probabilistic inference is exponential in the number of variables relevant to the computation; while the non-numeric computation in different heuristics is polynomial with respect to the number of variables related to the query [Li90]. The non-numeric computation can be very small if we randomly combine the whole distributions for a query and sum over those variables not queried. However, the total computational cost could be quite high in that case. It is important to realize that the role of non-numeric computation is to use its polynomial time cost effectively to reduce the exponential time cost of numeric computation. Therefore, when designing a probabilistic inference algorithm, the cost of non-numeric computation should be limited to a low degree polynomial. It should be clear that the maximum dimensionality of an algorithm for a query reflects the real computational complexity of a query in a belief network for a particular algorithm. The maximum

dimensionality will, in general, vary according to the algorithm used, for the same query in the same belief network. We are very interested in finding an algorithm which handles probabilistic inference in a belief network with the minimal maximum dimensionality.

Unnecessary computation. In many cases, not all nodes in a belief network are involved in a query. Including unnecessary nodes in a probability computation increases the total computation cost. It is important, for any algorithm, to consider only the relevant nodes of a query in probabilistic inference. In the symbolic probabilistic inference algorithm [D'A89, SDD90], a technique of dynamically determining the relevant nodes for a particular query is used. Though this technique may not be applicable to all algorithms, it can easily be applied to some algorithms.

Repeated computation. Since there could be multiple queries after some observations in a belief network, some of the computations may be calculated more than once. It is good for an algorithm to save some intermediate results for reuse in these kinds of queries. A caching strategy is employed for these tasks in the symbolic probabilistic inference algorithm [D'A89]. An experimental test of the algorithm has shown that the caching strategy could save about half of the computation time in the test cases in which the number of queries were randomly chosen from remaining variables after some observations [D'A89]. The caching strategy is closely related to the algorithm, and we will expand on this later in the thesis.

From the analysis above we know that there are three points we should consider to speed up a probabilistic inference algorithm:

1. Reduce maximum dimensionality in probabilistic inference as much as possible.

2. Eliminate unnecessary computation by considering only relevant nodes in probability computation.

3. Avoid repeated computation by using a caching strategy.

The first point above is the most important; it affects the computational complexity of probabilistic inference. The second point can be realized in polynomial time in the number of nodes of a belief network. Designing a caching strategy for avoiding repeated computation in an algorithm depends on the algorithm itself. In this thesis, we consider mainly the problem of how to reduce the maximum dimensionality in probabilistic inference in belief networks.

In the following chapters we will present a new approach for probabilistic inference in belief networks. It is a factoring approach that flexibly exploits the structure of a belief network and provides a mechanism for considering the above three points in a probabilistic inference algorithm.

Chapter 3

The Optimal Factoring Problem

3.1 An example

Even though the time complexity of computing the full joint probability of a belief network is invariant, the computational cost could vary depending on the order in which the probability distributions are combined. If the computational cost is very high, the variance can not be neglected. Let us consider the query of computing the full joint probability of a belief network with n nodes. It takes at least $(2^{n+1} - 4)$ multiplications to compute the probability if the number of values of each node is 2 and the graph formed by the n nodes is fully connected. The number of multiplications in the worst factoring case, compared with the best factoring result, is $(n-1)/2$ times more for the fully connected belief network.

As for the cases of querying subsets of nodes in belief networks, the variance of computational cost of different factorings is higher, because the time complexity could be different with different factorings. Therefore, the computational cost of probabilistic inference in a belief network depends on the factoring of combining the distributions of the relevant variables.

We give a simple example to show the effect of different factoring strategies on probabilistic inference in a belief network. Given the simple belief network in

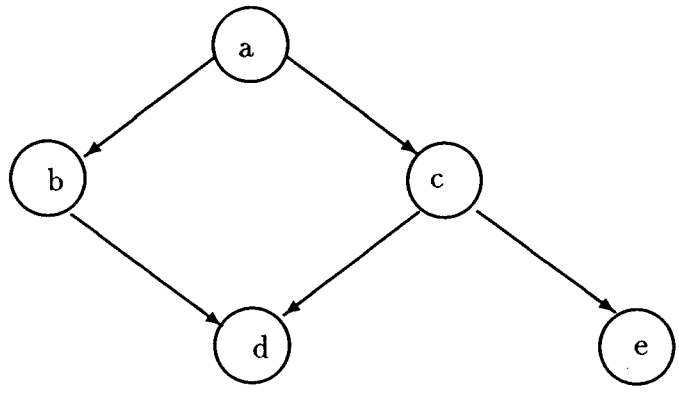


Figure 2. A simple belief network for two factorings.

figure 2, we want to query the joint probability of nodes d and e, namely $p(d, e)$. One factoring is given in the formula:

$$p(d,e) = [\sum_a [\sum_b [\sum_c [p(e|c)p(d|b,c)]p(c|a)]p(b|a)]p(a)]$$

which needs 72 multiplications. Another factoring needs only 28 multiplications:

$$p(d,e) = [\sum_c [p(e|c)[\sum_b p(d|b,c)[\sum_a p(c|a)[p(b|a)p(a)]]]]].$$

From this example we can see that different factoring gives significantly different computational costs.

In this chapter, we will formally define a combinatorial optimization problem, the optimal factoring problem. The purpose of proposing the optimal factoring problem is to apply some mature techniques developed for solving combinatorial optimization problems to the factoring problem and to utilize the results obtained from the optimal factoring problem for probabilistic inference.

3.2 The optimal factoring problem

An optimal factoring problem (OFP) with n expressions can be considered as a combinatorial optimization problem. Without loss of generality we assume that the domain size of each variable is 2. The problem can be described as follows.

Definition [FP] *Given*

1. *a set of m variables V,*

2. *a set of n subsets of V:* $S = \{S_{\{1\}}, S_{\{2\}}, ..., S_{\{n\}}\}$, *and*

3. $Q \subseteq V$, *a set of target variables*

define:

1. *combination of two subsets* S_I *and* S_J*:*

$$S_{I \cup J} = S_I \cup S_J - \{v : v \notin S_K \text{ for } K \cap I = \phi,$$

$$K \cap J = \phi, \text{ and } v \notin Q\},$$

$$I, J \subseteq \{1, 2, ..., n\},\ I \cap J = \phi;$$

2. *the cost function of combining the two subsets:*

$$\mu(S_{\{i\}}) = 0 \text{ for } 1 \leq i \leq n, \text{ and}$$

$$\mu(S_{I \cup J}) = \mu(S_I) + \mu(S_J) + 2^{|S_I \cup S_J|}.$$

$\mu(S_I)$ is not unique if $|I| > 2$. In general, it depends on how we combine the subsets. We indicate these alternative combinations by subscripting μ. $\mu_\alpha(S_I) = \mu$ shows the cost of combining result of S_I with respect to a specific tree structured combination of I, labeled α. We call this combination a factoring.

Definition [OFP] *The* **optimal factoring** *problem is to find a factoring* α *such that* $\mu_\alpha(S_{\{1,2,...,n\}})$ *is minimal.*

In above definitions, Q is a set of target variables after combining all subsets of S together; the set $\{v\}$ in the formula $S_{I \cup J}$ is the variables which do not appear in the remaining subsets of S after combination of S_I with S_J and not appear in the set Q. $\mu_\alpha(S_{I \cup J})$ is the total cost of combining all sets S_i ($i \in I, J$) in a given factoring order, and is determined by the dimensionality or the size of sets to be

combined and affected by the size of $\{v\}$ in previous combinations. If the domain size of each variable is not limited to 2, the value $2^{|S_I \cup S_J|}$ in above formula should be replaced with the multiplication of domain size of the variables in $S_I \cup S_J$. All possible factorings can be generated by permuting the n subsets S_i and then putting parentheses in all rational ways in the permutation to form all $S_{\{1,2,\ldots,n\}}$.[6]

An instance of OFP can be represented by a network. A network for the instance is a fully connected graph with n nodes corresponding to the n sets $S_{\{1\}}$ to $S_{\{n\}}$. If a weight is assigned to each path between two nodes, the weight of the path from $S_{\{i\}}$ to $S_{\{j\}}$ in the network is $|(S_{\{i,j\}})|$. Two values, b_i and d_i, are attached to each path and are dynamically determined. b_i, the reduced cardinality, is the number of variables to be removed from node $S_{\{i\}}$ to node $S_{\{j\}}$, which is determined by the path from the start node to node $S_{\{j\}}$. d_i, the increased cardinality, is the number of variables which appear in the result from start node to node $S_{\{j\}}$ but not in $S_{\{i\}} \cup S_{\{j\}}$. b_i and d_i may have different values if the path is from $S_{\{j\}}$ to $S_{\{i\}}$. The total weight between two nodes can be calculated by adding the weight on the edge with d_i together and subtracting b_i. Also, we can consider the value of exponent of the total weight as the distance between any two nodes. If we ignore the insertion of parentheses among all nodes ordered by a path, the OFP is the problem of finding a way of passing through each node exactly once in the network with the shortest distance.

The OFP generally is a hard problem. We guess that OFP is an NP-hard problem even though we have not yet proven it. We can see the similarity between OFP and the problem of finding the shortest path among n nodes by passing each node exactly once (SPP) [Law76], which is NP-hard. In SPP, the problem is to find a permutation of n nodes which results in the shortest path; while in OFP, the problem is to find a proper permutation of n nodes and then put parentheses in so that it results in a minimal computation. If we ignore the parentheses in the result

[6]Strictly speaking, there are many duplicates generated in this way. For example,,$((ab)c)$ is the same factoring as $(c(ab))$, where a, b, and c denote factors.

of OFP, since the time complexity of putting parentheses in a given permutation of the n nodes to get an optimal result is polynomial in the number of the nodes [Hu82], OFP like SPP is the problem of finding a proper permutation of n nodes. The difference between the two problems is that in SPP edge distance of two nodes are static; while in OFP, they are dynamic, that is, they depend on the path taken to the edge.

3.3 Some results for the OFP

Although the OFP generally is a hard problem, some restricted instances of the OFP have polynomial time algorithms. For example, given a domain of variables, if each pair of sets S_i and S_j is disjoint and the set Q is the union of all the sets S_i, then the optimal ordering of $\alpha(S_{\{i_1,...,i_n\}})$ can be obtained in linear time. In this section, we will explore factoring methods for particular instances of the OFP. These factoring methods help us to find efficient probabilistic inference algorithms. We will also present an optimal factoring algorithm for an arbitrary belief network.

Lemma 3.1 *Given a factoring problem with n variables* $\{1,2,...,n\}$*:* $S_1 = \{1\}$, $S_2 = \{2\}$,..., $S_{n-1} = \{n-1\}$, $S_n = \{n\}$ *and* $Q = \{1,2,...,n\}$, *one of the optimal factorings is to combine any int((n+1)/2) single variable factors, called marginals, first, then to combine the rest of the single variable factors together, and finally to combine the two results.*

Proof: We prove the lemma by induction. Given n=2, there is only one possible combination. If n=3, any two marginals can be combined first, then the result will be combined with the other marginal. The order of combination meets the order described in the lemma and is optimal. Assume that the combination order in the lemma is optimal for n less than or equal to k marginals. In the case n=(k+1), the result of combining (k+1) marginals must result from the combination of combining k combined marginals with one marginal, or combining (k-1) combined marginals

with 2 combined marginals, and so on. Remember that the cost function defined in the definition OFP is

$$\mu(S_{I\cup J}) = \mu(S_I) + \mu(S_J) + 2^{|S_I \cup S_J|}, \tag{3.2}$$

that is, the cost for the final step is 2^{k+1} which is independent of the distributions of the two factors to be combined. We must prove that the combination order in the lemma minimizes $\mu(S_I) + \mu(S_J)$. If we use a number to denote the size of a set, then we need to prove that given $k < m$,

$$\mu(m+1) + \mu(k) > \mu(m) + \mu(k+1). \tag{3.3}$$

According to the cost function (3.2), there exist m_1, m_2, k_1 and k_2 which meet the formulas: $m_1 \geq m_2$, $k_1 \geq k_2$, $m_1 + m_2 = m + 1$ and $k_1 + k_2 = k$ such that $\mu(m+1)$ and $\mu(k)$ are both optimal in the left side of formula (3.3). If we choose the decomposition for the right side of formula (3.3) relevant to m_1, m_2, k_1 and $k2$, then to prove formula (3.3), we need to prove

$$\mu(m_1) + \mu(m_2) + 2^{m+1} + \mu(k_1) + \mu(k_2) + 2^k \tag{3.4}$$

is greater than

$$\mu(m_1 - 1) + \mu(m_2) + 2^m + \mu(k_1) + \mu(k_2 + 1) + 2^{k+1}. \tag{3.5}$$

From (3.4) and (3.5) we get

$$\mu(m_1) + 2^{m+1} + \mu(k_2) + 2^k > \mu(m_1 - 1) + 2^m + \mu(k_2 + 1) + 2^{k+1}. \tag{3.6}$$

Since $2^{m+1} \geq 2^m + 2^{k+1}$, it is sufficient to prove the following formula instead of the formula (3.6)

$$\mu(m_1) + \mu(k_2) + 2^k > \mu(m_1 - 1) + \mu(k_2 + 1).$$

If we decompose $k_2 + 1$ into two factors with number k and 1, then the formula is

$$\mu(m_1) + \mu(k_2) + 2^k > \mu(m_1 - 1) + \mu(k_2) + \mu(1) + 2^{k_2+1},$$

that is

$$\mu(m_1) + 2^k > \mu(m_1 - 1) + 2^{k_2+1}.$$

Then we should prove the following formula since $k \geq k_2 + 1$:

$$\mu(m_1) > \mu(m_1 - 1). \tag{3.7}$$

The correctness of formula (3.7) is obvious for marginals. Thus we have proven formula (3.3). From formula (3.3) we know that if m+k+1 is even, the minimal value results from the decomposition into two sets with equal size; and if m+k+1 is odd, the minimal value results from the decomposition in which one set has one more factor than the other set. This meets the combination order in the lemma. For the two decomposed sets, they both have less than k marginals and can be combined optimally according to the induction assumption. ∎

Lemma 3.2 *Given a factoring problem with n variables* $\{1,2,...,n\}$*:* $S_1 = \{1\}$, $S_2 = \{2\}$,..., $S_{n-1} = \{n-1\}$, $S_n = \{1,2,...,n\}$ *and* $Q = \{n\}$, *the optimal factoring is to combine any int((n+1)/2) single variable factors according to lemma 3.1, then to combine the result with the factor* S_n. *The original factoring problem then becomes a new factoring problem with factors* S_{k_1}, S_{k_2}, ..., S_{k_i}, $S_{k_{i+1}} = \{k_1,...,k_i,n\}$ *and* $Q = \{n\}$, *where i=n-int((n+1)/2), which has the same style as the original problem. The same strategy can then be used for the problem until a final result is obtained.*

Proof: We prove the lemma by induction. For n=2, the combination is unique. For n=3, according to the lemma, we combine two marginals first and then combine the result with the conditional factor. The cost is $2^2 + 2^3$ and is minimum. Assume that the combination order in the lemma is optimal for n less than or equal to k. Then we will prove the combination order is also optimal in the case n=k+1. Some notation must be introduced first. If the number of multiplications for combining m marginals, in accordance with lemma 3.1, is denoted as $M(m)$, then $M(1) = 0$,

and $M(m)$, for $(m > 0)$, can be recursively computed as

$$M(m) = 2^m + M(int(m/2)) + M(m - int(m/2)). \tag{3.8}$$

There is a combination order for S_1, S_2,..., S_n and m=int(n/2) such that the number of multiplications for combining

$$((S_n(...(S_1S_2)...S_m))(...(S_{m+1}S_{m+2})...S_{n-1}))$$

is

$$2^n + (2^2 + ... + 2^m) + 2^{n-m} + (2^2 + ... + 2^{n-m-1}). \tag{3.9}$$

We know that the total number of combinations needed for computing S_1, S_2,..., S_n is $(n-1)$ and the number of multiplications needed for combining all factors in the worst case is

$$2^2 + 2^3 + ... + 2^{n-1} = 2^n - 4. \tag{3.10}$$

If we denote $F(n)$ as the number of multiplications needed for combining S_1, S_2,..., S_n then $F(n)$ can be represented as follows if m marginals are combined first:

$$F(n) = 2^n + M(m) + F(n-m). \tag{3.11}$$

Then, proving the combination order for combining S_1,..., S_n is equal to proving the following inequality. Given s and t, $s \neq t$ and t is the value chosen as in the lemma, then

$$2^n + M(s) + F(n-s) > 2^n + M(t) + F(n-t). \tag{3.12}$$

First we consider the case $n = 2t$ and $s < t$, and assume $s = t - j$ for $1 \leq j \leq t$, then we prove the following formula with the s and t:

$$M(t-j) + F(n-t+j) > M(t) + F(n-t). \tag{3.13}$$

From formulas (3.8) and (3.9) we know that the dominant terms in the left side of formula (3.13) are $2^{t+j} + 2^{t-j} + ...$ The rest of the terms are much smaller compared with the two dominant terms. The dominant terms in the right side of

the formula are $2^t + 2^t + ...$, and the rest of the terms are also much smaller than the dominant terms. According to formula (3.10), we know that formula (3.13) is true for $s = t - j$, for $1 \leq j \leq t$.

Next we prove the case $n = 2t$ and $s > t$. We consider the case $s = t + j$ for $1 \leq j \leq t$. Given s and t, we should prove

$$M(t+1) + F(n-t-1) > M(t) + F(n-t). \tag{3.14}$$

From formulas (3.8) and (3.9) we know that the dominant terms in the left side of formula (3.14) are $2^{t+j} + 2^{t-j} + ...$, and the rest of the terms are much smaller than the dominant terms. Similarly, the dominant terms on the right side of the formula are $2^t + 2^t + ...$ and the rest of the terms can be ignored in comparison with the dominant terms. This tells us that formula (3.14) is correct for $s = t + j$ for $1 \leq j \leq t$.

Similarly we can prove formula (3.12) in the case $n = 2t-1$ for $s < t$ or $s > t$. For $s > t$ the proof is similar to the above proof. But for $s < t$, if we substitute s in formula (3.12) with $s = t-1$, both sides of the formula are equal. If we use $s < t-1$ instead of $s = t - 1$, formula (3.12) is true. This means the optimal combination is not unique in this case. For example, if $n = 5$ in the factoring problem, the combinations $((S_5(S_1S_2))(S_3S_4))$ and $((S_5((S_1S_2)S_3))S_4)$ have the same result.

According to formula (3.12), we combine t marginals first. t is determined as above. Then combine the result with the factor S_n. After the combinations the number of marginals left is less than k and they can be combined optimally according to the induction assumption. Thus the combination order for S_1, S_2,..., S_n is proved. ∎

Lemma 3.3 *Given a factoring problem with n variables* $\{1, 2, ..., n\}$*:* $S_1 = \{1\}$, $S_2 = \{2\}$,..., $S_n = \{n\}$, $S_{n+1} = \{1, 2, ..., n\}$ *and* $Q = \{n\}$, *the optimal factoring is to combine* S_1,..., S_{n-1} *with* S_{n+1} *first, then combine the result with* S_n. *The order of combining* S_1,..., S_{n-1} *with* S_{n+1} *is given in lemma 3.2.*

Proof: From lemma 3.2 we can see that to combine S_1,..., S_{n-1} with S_{n+1} is the same factoring problem described in lemma 3.2, so the factoring, according to the lemma, is optimal. The result of the combination is a set with one variable in it: $\{n\}$. Combining the result with S_n, the dimensionality of the combination is just 1. So, the combination is minimal for any combination of two factors. On the other hand, if we exchange the combination of any S_i $(1 \leq i < n)$ with S_n, the result of combining $(n-1)$ marginals with S_{n+1} has a dimensionality of 2 and the dimensionality of combining the result with S_n is 2 also. Then the cost of this combination order is bigger than the cost of the given combination in the lemma. Therefore, the combinations given in lemma 3.3 is optimal. ∎

Lemma 3.4 *Given a factoring problem with n+k-1 sets on n variables* $\{1,2,...,n\}$*:* $S_2 = \{2\}$,..., $S_{n-1} = \{n-1\}$, $S_n = \{1,2,...,n\}$, k $\{1\}$*'s (we may denote them as* $S_{1,1} = \{1\}$,... $S_{1,k} = \{1\}$*), and* $Q = \{n\}$*, one of the optimal factorings is to combine the k* $\{1\}$*s first, then optimally combine the result with the remaining factors according to the lemma 3.3.*

Proof: It is obvious that combining the k $S_{1,i} = \{1\}$s $(1 \leq i \leq k)$ together first is optimal. Combining the result with the remaining factors is optimal according to the lemma 3.3. Therefore, the factoring in the lemma is optimal. ∎

Lemma 3.5 *Given a factoring problem with n variables* $\{1,2,...,n\}$*:* $S_1 = \{1\}$, $S_2 = \{1,2\}$, $S_3 = \{2,3\}$..., $S_n = \{n-1,n\}$*, and* $Q = \{n\}$*, then the optimal factoring is to combine these factors in the ascending order of their subscripts. That is, combine the* S_1 *with* S_2 *first, then combine the result with* S_3*, and so on.*

Proof: From the definition of FP we can see that the dimensionality of each combination, as specified by the lemma, is 2 and that one variable is removed from the result after each combination. Since the size of each S_i for $i > 1$ is equal to 2, every combination step must have a dimensionality of at least 2. Therefore the given combination is minimal. Therefore, the factoring is optimal. ∎

For the arbitrary factoring problem, we have developed an optimal dynamic factoring algorithm. Dynamic programming is one of the few general techniques for solving optimization problems [NW88, PS82, Hu82]. It is related to branch-and-bound techniques in the sense that it performs an intelligent enumeration of all feasible points of a problem, yet in a different way. The idea is to work backwards from the last decisions to the earlier ones. Using the dynamic programming approach to OFP, we start backwards from an assumed optimal result. According to the "principle of optimality", any sub-combination of n factors must be an optimum itself, and all possible sub-combinations may be used in the final optimal result. We keep all computed optimal sub-combinations, and use tables to save all intermediate results. Thus the dynamic programming approach for OFP can be described as:

1. Generate all combination tables from 1 to n. The ith combination table can be generated from all pairs of combination tables (j, k) such that $j + k = i$. The elements (combined factors) chosen from the jth and kth tables must be exclusive. For the combinations having the same elements only that one which has the minimal number of multiplications is saved in the table for subsequent use.

2. An optimal combination is any entry in the nth combination table with the lowest number of total multiplications needed.

The dynamic approach will find an optimal result, but depends on comparing all possible factoring results in each factoring step to get the best one. It can be seen that if a kind of best-first search is applied to find a best result, the time complexity of the algorithm for computing the $(n+1)th$ table would be $O(n^2 * 2^n)$ in the number of factors. In the ith combination table there are $(n!/(i!(n-i)!))$ elements since only one combination of any i elements is a candidate for the $(i+1)th$ combination. The number of elements in the ith table is the number of combinations of choosing i elements from n, so there are a total of 2^n elements in all n tables. Since there are $n!$

distinct factoring results for n factors, the dynamic programming approach results in substantial savings. Even though the dynamic strategy is useless in practice, it is useful in research as an analytical tool to check how close an approximation algorithm is to an optimal result.

Since the general OFP appears to be a hard problem, we must search for approximation methods and heuristics, or identify special cases for which efficient algorithms exist. Two criteria for a heuristic strategy are quality, i.e. the closeness of the result of a heuristic to an optimal result, and the time complexity of the heuristic algorithm itself. There is a trade-off between the quality and time complexity in a heuristic algorithm. The following are some possible heuristic greedy strategies:

1. In each step of choosing a pair of factors to combine, we may consider the pair of factors which gives the minimum μ value as a candidate for combination;

2. In each step, we may consider the pair which has the smallest dimensional result as a candidate for combination.

We will see later that the strategy of taking the pair with the smallest dimensional result as a candidate shows good results in the application of probabilistic inference in belief networks. Considering the similarity between SPP and OFP, we may explore the heuristic methods used for SPP to OFP, for example, we may use the 'nearest neighbor' strategy in OFP.

3.4 Mapping between OFP and probabilistic inference

Our interest is in the application of OFP to probabilistic inference. The direct use of OFP is to find an optimal evaluation tree for computing queries in a belief network. Given a belief network with m nodes and a set of observations, a query involves identification of a subset of n nodes relevant to the query, and computation of the conformal product [SDD90] of marginal and conditional probabilities of the n nodes. The n nodes with their relations can be mapped to the symbols in the

definition of OFP: the n nodes with their immediate antecedent nodes are mapped to the n subsets of m variables; the queried nodes correspond to the variables in the subset Q; $S_{I \cup J}$ denotes the intermediate result of conformal product of distributions I and J; and μ gives the number of multiplications needed for the computation. Finding an optimal factoring minimizes the number of multiplications needed for this computation. From the definition of OFP we can see that two values influence the result of a factoring. One is the dimensionality of the union of two sets S_I and S_J which directly determines the current combination cost. The other is the number of variables to be removed from a combined result which affects the dimensionality of subsequent combinations. The two factors are related and should be considered together in finding an optimal factoring result. It should be clear that determining the lowest cost way to combine distributions in a belief network is an OFP.

We give a simple example to show the mapping between OFP and probabilistic inference. In figure 2, we want to compute the joint probability $p(d, e)$. The mapping is as follows. $S_1 = \{a\}$, $S_2 = \{a, b\}$, $S_3 = \{a, c\}$, $S_4 = \{b, c, d\}$, $S_5 = \{c, e\}$, $Q_1 = \{d, e\}$. If $\alpha = ((((S_1 S_2) S_3) S_4) S_5)$, then $\mu_\alpha(S_{1,2,3,4,5}) = 28$ (see the example in section 3.1).

From the OFP point of view, we can see that previously developed exact probabilistic inference algorithms are just different factoring strategies. However, since these factoring strategies are constrained by the structure of the original graph or a derived graph, it is hard for these strategies to find optimal factorings, or even to provide a measure for the optimality of the strategies. So, their performances are not as good as the performance resulting from a strategy from the OFP point of view (we will present a factoring strategy later). Some of the strategies could be improved by considering optimization in their algorithms. In the next chapter we will discuss the optimal factoring problem in probabilistic inference in a belief network.

Chapter 4

Optimal Factoring for Probabilistic Inference

4.1 Optimal factoring for singly-connected belief networks

From chapter 3 we know that finding an optimal factoring in general is a hard problem. That is, we don't expect to find an efficient optimal factoring algorithm for an arbitrary belief network in probabilistic inference. However, there exists a polynomial time algorithm for generating optimal factoring for tree-structured (including poly-tree) belief networks. In this section, we will present the algorithm. The optimal factoring algorithm is based on the lemmas in chapter 3.

The meaning of lemma 3.2 to lemma 3.5 of chapter 3, corresponding to probabilistic inference in belief networks, can be explained from very simple belief networks. Lemma 3.2 can be explained from figure 3(a), in which the nth variable is queried and the rest of the variables are marginals. The lemma tells us an optimal factoring strategy to compute the marginal probability of the nth variable. Lemma 3.3 also refers to the graph, which is used for querying the conditional probability of $p(n|n+1)$ if node $(n+1)$ is the child of node n and is observed. Lemma 3.4 describes a more general case for figure 3(c). A simpler query for the graph is $p(1|2,3,...,n)$ where node i $(i > 1)$ is observed. Lemma 3.5 refers to a belief network with chain structure (see the figure 3(b)) in which the marginal probability of node n or the marginal probability of node 1, given observation of node n, is

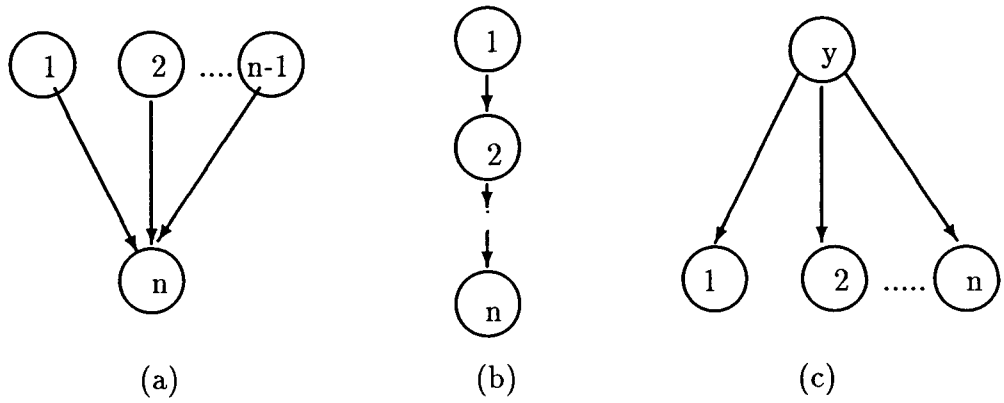


Figure 3. Different cases of query in poly-trees.

queried. The lemma tells us the fact that the cost of combining two non-marginal nodes, which are not directly connected, is always bigger than the cost of combining two nodes which are directly connected. The networks in figure 3 represent the basic structures for decomposing a singly-connected belief network.

We introduce some new names for the purpose of easy description in the rest of the section. We call a node with its parents a *group* and the node itself group *head*; and a marginal node is the only node in a group and is the group head.

Theorem 4.1 *There exists a linear time algorithm to generate an optimal factoring for querying the marginal probability of a node in a poly-tree.*

Proof: Based on the factoring strategies in lemma 3.2 to lemma 3.5, we can construct an optimal factoring strategy for a poly-tree. Given some observed nodes and a queried node in a poly-tree, the nodes relevant to the query still form a poly-tree. The nodes that are the antecedents of the queried node, in the original poly-tree, are in the formed poly-tree and the descendants of those antecedents must be observed nodes or antecedents of some observed nodes. A queried node divides all nodes of the formed poly-tree into two parts: the nodes connected to it from its parents, and the nodes connected to it from its children. The optimal factoring strategy starts factoring from the queried node and spreads out to the whole tree.

Two operations used in the factoring strategy are defined as follows:

1. (Bottom-up) In computing the marginal probability of a group head, if some other nodes in the group have unknown marginal probabilities, those groups with an unknown marginal probability node as group head should be computed first.

2. (Top-down) In computing the marginal probability of a group head, if the head has any child, then the group with the child as the head should be computed first and the target variable, i.e. the result variable after combining a group together, is the head of the first group.

The factoring strategy is the following. Compute the probability of the queried node from the group in which the queried node is a head. If any node in the group has unknown marginal probability, then apply the bottom-up operation. And if the queried node has any child node, then apply the top-down operation. The top-down and bottom-up operations are repeatedly used for any group wherever they are applicable, but not to one node repeatedly in order to avoid an infinite loop. If no more bottom-up and top-down operations are needed in a group, use lemmas 3.2 or 3.3 to compute the target variable of the group. If some computed group has the form in figure 3(c), then apply lemma 3.4 to combine the nodes.

Since there is one node to be combined each time using the top-down or bottom-up operation, the factoring is linear in the number of nodes relevant to the query.

The optimality of the factoring strategy can be illustrated as follows. First we see that the factoring within any group is optimal, i.e., all groups formed in the factoring strategy have forms in lemmas 3.2 or 3.4 (hereafter we call them form 1 and form 2 respectively), or can be converted to one of the forms. If the group with the queried node as a head can not be computed in one of the forms, we use top-down and/or bottom-up operations to generate new groups. By repeatedly using the bottom-up operation we will meet some groups in form 1 style, since each root

node is either a marginal node, an observed node, or the queried node in the formed poly-tree. After these groups have been computed, those groups which contain the head of the just-computed groups as member have known marginal probabilities of all non-head nodes and they either become form 1 style or need top-down operation. If some of them are form 1 style, they can be computed again, and so on for the other groups.

The groups generated from top-down operation are either form 2 style or need more bottom-up and/or top-down operations to generate new groups. The groups generated by the bottom-up operation have form 1 style as described above. Those groups generated by repeatedly using top-down operation must be form 2 style because a leaf in the formed poly-tree is an observed node. By applying lemma 3.4 to these groups, we can compute the values needed to return to the group head that generated the computed groups using top-down operation. A node may have more than one returned value, depending on the number of its children. All values returned to one node can be multiplied together as a new value to return to the node according to lemma 3.4. The group having a returned value then becomes form 2 style. Notice that we take the group as form 1 style here because the group with a returned value can be computed in lemma 3.3. This process can be repeated until a value returns to the queried node.

Second we see that each group generated by using top-down and/or bottom-up operations can be computed optimally according to lemma 3.5. This can easily be shown by induction on the number of groups in the poly-tree. Therefore, the optimality of the factoring strategy is ensured. ∎

In probability computation, any computation result within a group or among groups can be cached for subsequent use. The top-down and/or bottom-up operations will be avoided if there are cached intermediate results available.

From the combinatorial optimization point of view we know that the poly-tree propagation algorithm [KP83, Pea88] and Revised poly-tree Algorithm [Peo91]

provide an optimal factoring among groups for computing probabilities; but their propagation strategies do not provide any factoring strategy within a group.

4.2 Factoring in multiply-connected belief networks

We have shown that there is a polynomial time optimal factoring algorithm for poly-trees and there is an exponential time optimal factoring algorithm for arbitrary belief networks. We doubt that there exists a polynomial time optimal factoring algorithm for an arbitrary belief network because we believe that OFP is an NP-hard problem. In this section we will list some factoring strategies used for multiply-connected belief networks and discuss their advantages and disadvantages.

4.2.1 Examples of factoring strategies

From the optimal factoring point of view, previously developed algorithms are just different factoring strategies. If we separated non-numeric computation from numeric computation or conformal products and postponed all numeric computation, the factoring strategies used in those algorithms would generate different factorings for any query in probabilistic inference. We will examine the factoring strategies of two popular probabilistic inference algorithms, the clustering algorithm [Pea88] and the symbolic probabilistic inference algorithm (SPI) [D'A89, SDD90].

First we look at the clustering algorithm. There are basically two factoring strategies used in the algorithm. The first factoring strategy is to create cliques. The work of converting a belief network into a Markov network and triangulating the network to form cliques determines the factoring for combining nodes locally. The second factoring strategy is to assemble the cliques into a join tree (or junction tree in some other clustering algorithms). The poly-tree propagation algorithm on join tree [Pea88] gives the factoring among these clique nodes. A feature of the factoring part of the clustering algorithm is that the factoring is accomplished for a belief network before any query and observation.

The factoring strategy in the symbolic probabilistic inference algorithm is in two parts. The first task is to create a partition tree from the original belief network. The second task is, given a query, to start symbolic inference from the root of the partition tree and determine the nodes relevant to the query and the way of combining them. In more detail, the tasks are accomplished by three factoring steps. Generating a partition tree is the first factoring step. This step partitions a set of nodes and arranges the partitions into a tree. The local ordering heuristic is the second factoring step in the algorithm which gives the combining order of nodes within a partition. And finally, control strategy, another factoring step, determines the combination ordering among partitions. Therefore, three components in SPI - partitioning, local ordering heuristic and control strategy - determine the final factoring.

Besides the functional difference of factoring strategies in these examples, an obvious difference between these factoring strategies is that the symbolic probabilistic inference algorithm provides one more factoring step within a partition. A most important feature of the SPI factoring strategy which differs from that of the clustering algorithm is that the second and third factoring steps are carried out after a query. An experimental test [Li90] has shown the significance of this query-driven factoring in improving efficiency of probabilistic inference.

A common factoring technique used in the above algorithms is that a local factoring is imposed on the belief network through a tree created from the original belief network, an undirected tree in the clustering algorithm, and a directed tree in SPI. Structures of trees and ways of creating trees are important to the efficiency of these algorithms. Some efforts have been made for finding optimum triangulation in clustering algorithm [Wen90] and finding a better partition tree in the SPI algorithm [SDD90, D'A90]. The issue about the advantages and disadvantages of the factoring techniques will be discussed in the next sub-section.

The factoring steps used with the OFP method can typically be described as follows[7]. Given a belief network with observations and a query, a set of nodes relevant to the query is found; then a factoring strategy is applied to these nodes, and their distributions, to get a factored query expression. The probability computation is carried out based on the factored expression (or evaluation tree). The factoring step and the probability computation can be mixed together if needed. The key difference of the method is that the evaluation tree is dynamically determined. The markable advantage of this factoring method is that it provides, quantitatively, measurements of factoring results. The comparison of these method with other probabilistic inference algorithms or methods will be discussed in chapter 7.

4.2.2 Static factoring vs. dynamic factoring

Factoring strategies can be *static* – used before any query – or *dynamic* - used just after each query but before real probability computation. In this sub-section, we will discuss the advantages and disadvantages in static factoring strategy and dynamic factoring strategy for probabilistic inference in a belief network.

In static factoring, the order of combining factors comes from the original belief network before any querying and observation. An example of a static strategy is the partition strategy in SPI [D'A89], which creates a partition tree before any probability computation. One of the advantages of static factoring is that it is performed only once before any querying and observation, and can be performed off line; the other advantages are that the inference algorithm with a static factoring strategy is usually incremental with respect to both queries and observations, and intermediate results can be cached for later use. The disadvantage of static factoring is that it imposes some constraints on the ordering of combining some distributions without considering the effect of observations and querying tasks. Since the graphs corresponding to different queries with different observations

[7]It should be noted that some ideas used in some papers, like factorization in the paper [SS90] and factoring in paper [SDD90], are different from the optimal factoring problem defined here.

are very different given a belief network, the constraints may be far from optimal in some queries.

Dynamic factoring will find a combination order of factors according to the factors themselves, and only the factors relevant to the current query, not to the original belief network, are considered. The local ordering heuristic in SPI is an example of dynamic factoring. Another example of dynamic factoring is in generating a partition tree. The partition tree will be generated after each query, given some observations, and only those nodes relevant to the current query are in the partition tree. The merit of dynamic factoring is that it may find a better factoring result than a static factoring strategy does because it has more information available, namely the specific query to be answered. The drawbacks of dynamic factoring are as follows. First, it runs every time after each query; and second, caching becomes difficult. Considering that a polynomial time cost for a factoring strategy will reduce exponential time cost for probability computation, we believe it is worthwhile to use this strategy for every query. This is the idea of trading non-numerical computation for numerical computation in probabilistic inference.

Dynamic factoring may be divided into a control strategy and a local factoring method. The control strategy, usually considering global information, controls the data flow of computation; and local factoring, usually considering local information, determines local evaluation. Consider the optimal factoring algorithm for poly-trees described in the previous section as an example. In that algorithm, the control strategy is the bottom-up and top-down operations and local factoring is the combination ordering given by lemmas 3.2 and 3.3. Another example is the implementation of the control heuristic and local ordering heuristic in SPI. The control heuristic is a recursive decomposition of queries from the root of a partition tree; the local ordering heuristic determines the combination of terms in each decomposed level. The control strategy and local factoring may be mixed together as the implementation in set-factoring (presented in the next section), but a separated

strategy may be more flexible and efficient since it considers both global and local information.

Considering the advantages of static factoring and dynamic factoring, a better factoring algorithm may result from mixing the two strategies in one algorithm. SPI is an example of the combination of the two strategies. There is a trade-off between the factoring time and the factoring result since an optimal result for any belief network could be obtained after considering many factoring features, like the dynamic programming strategy given in chapter 3, but it takes exponential time.

Local factoring is a comparatively easier problem than the control strategy problem in dynamic factoring since local factoring uses local information only. The control strategy is a key point in factoring and is a hard problem.

4.2.3 Caching strategy

One possible difference between static factoring and dynamic factoring is the reuseability of previous factoring structure or intermediate results in a multiple query situation. This problem is closely related to the caching strategy used in a factoring algorithm.

Caching may reduce probabilistic computation depending on the structure of a belief network and the tasks to be carried out, as the test results indicated in the papers [D'A89, SDD90]. Some tasks favor caching: for example, given a set of observations in a belief network and a set of queries on more than one variable. Some belief networks provide good caching structures: for example, a belief network having long chains would provide many opportunities for caching when queried nodes are all in the chain. A caching strategy is directly related to a factoring strategy; namely, a cached result in one factoring algorithm may not be used in another factoring algorithm. The number of intermediate results cached for subsequent use is inversely proportional to the work to get future results.

An experimental test has been performed for examining the effects of caching between the SPI algorithm [D'A89], with a static factoring strategy for a partition

tree, and the set-factoring algorithm with a dynamic factoring strategy for creating an evaluation tree (see next section). The test cases were a set of randomly generated belief networks with tasks of querying marginal probabilities for a subset of the variables. Each queried variable is randomly chosen from the remaining variables after inserting some observations randomly in each belief network. The experiment has shown that the effect of the caching strategy for the set-factoring algorithm is significant and is comparable to that of the caching strategy for the static factoring algorithm (SPI). Even though the effects of caching in our test depends on the factoring strategy and caching strategy, these results indicate caching is useful in dynamic factoring algorithms.

Another difference between the static factoring and dynamic factoring is that a static factoring can be compiled off-line for fast response to a query. If a dynamic factoring can efficiently generate an evaluation tree and has a good trade-off between the polynomial cost of non-numeric computation and the exponential cost of numeric computation, then the overhead of dynamic factoring will be low.

4.3 A heuristic factoring algorithm

In chapter 2, we have summarized three points for reducing computational cost in probabilistic inference in belief networks – minimizing the maximum dimensionality of a query, avoiding unnecessary computation, and reducing repeated computation. We are mostly interested in the first point: minimization of the maximum dimensionality. We do not address the second point because there is a way to find relevant nodes for a query in polynomial time in the number of nodes in the belief network[SDD90] and we can directly use those nodes in the factoring problem. As for the third point, it is constrained by the tasks; we will not discuss the issue here.

The problem of minimizing the maximum dimensionality for a query is not exactly the OFP. A factoring with minimized dimensionality for a query may not be optimal, while an optimal factoring result will usually have minimal dimensionality.

If we consider the optimal factoring problem as a search over the entire factoring space, the optimal factoring problem may be more difficult than the problem of minimizing the maximum dimensionality. In the search for optimal factoring, we have to find a sub-optimal result in each step; namely, consider global information for the best result. In the search for the factoring with the minimal maximum dimensionality, we may use some heuristic strategy locally to obtain the result. We will see later that the algorithm, set-factoring, we have developed for the factoring problem is in fact partly for finding a minimal maximum dimensionality. The probability computation cost based on the factorings from minimizing the maximum dimensionality and OFP should be at worst a linear function of the number of nodes relevant to the query. That is, the problem of minimizing the maximum dimensionality is as hard as OFP.

4.3.1 The set-factoring algorithm

We now present an efficient heuristic algorithm, called set-factoring, we have developed for finding good factorings for probability computation. In a belief network with nodes $\{x_1, x_2, .., x_n\}$ connected by arcs, the general form of a query is $P(X_J|X_K, X_E)$, where X_J is a set of nodes being queried, X_K is a set of conditioning nodes and X_E is a set of observed nodes. $P(X_J|X_K, X_E)$ can be computed from $P(X_J, X_K|X_E)$. For simplicity, we will only consider the case $P(X_J|X_E)$ in the algorithm. This ignores several potential simplifications noted in [SDD90], but simplifies the presentation.

Given a query $P(X_J|X_E)$ in a belief network, often only a subset of the nodes is involved in the probability computation. The involved nodes can be chosen from the original belief network by an algorithm which runs in linear time in the number of nodes and arcs in the belief network [GVP89]. Once we have obtained the nodes needed for the query, we have all the factors to be combined. In accordance with the definition 3.2, we have n subsets of n nodes and set Q. We use the following algorithm to combine these factors.

1. Construct a *factor set* A which contains all factors to be chosen for the next combination. Each factor in set A consists of a set of nodes. Initialize a *combination candidate set* B empty.

2. Add all pairwise combinations of factors of the factor set A to B if the combination is not in set B, except the combination of two factors in which each factor is a marginal node and they have no common child; and compute the $u = (x \cup y)$ and sum(u) of each pair. Where x and y are factors in the set A, sum(u) is the number of nodes in u which can be summed over when the probability computation corresponding to the two factors is carried out.

3. Choose elements from set B such that C=$\{u|u : minimum_B(|u| - sum(u))\}$, here $|u|$ is the size of u excluding observed nodes. If $|C| = 1$, x and y are the factors for the next combination; otherwise, choose elements from C such that D=$\{u|u : maximum_C(|x| + |y|),\ x, y \in u\}$. If |D| = 1, x and y are the terms for the next multiplication, otherwise, choose any one of D.

4. Generate a new factor by combining the candidate pair chosen from the above steps and modify the factor set A by deleting two factors of the candidate pair from the factor set and putting the new factor in the set.

5. Delete any pair of set B which has non-empty intersection with the candidate pair.

6. Repeat step 2 to 5 until only one element is left in the factor set A which is the final combination.

Following is an example to illustrate the algorithm by using the network shown in figure 4. Suppose that we want to compute the query $p(4)$ for the belief network and assume that there are 2 possible values of each variable. The nodes relevant to the query are {1, 2, 3, 4}. We use the set-factoring algorithm to get their combination.

Loop1:

Factor set A is {1,2,3,4}.

The set B is {(1,2) (1,3) (1,4) (2,3) (2,4) (3,4)} after step 2.

The current combination is (1,2), i.e. $p(2|1)*p(1)$ after step 3 (there was more than one candidate in this step, we chose one arbitrarily).

The set A is {(1,2), 3, 4} after step 4.

And the set B is {(3,4)} after step 5.

Loop2:

Factor set A is {(1,2),3,4}.

The set B is {((1,2),3) ((1,2),4)) (3,4)} after step 2.

The current combination is ((1,2),3) after step 3.

The set A is {(1,2,3), 4} after step 4.

And the set B is empty after step 5.

Loop3:

Factor set A is {(1,2,3),4}.

The set B is {((1,2,3),4)} after step 2.

The current combination is ((1,2,3),4) after step 3.

The set A is {(1,2,3,4)} after step 4.

And the set B is empty after step 5. The factoring result is

$$p(4) = \sum_{2,3}[p(4|2,3)[\sum_{1}[p(3|1)[p(2|1)p(1)]]]].$$

There are several things that should be noticed in the algorithm. First, queried nodes should not be deleted from any terms in the expression, and if a node is a queried node and it has no parents, then the node will be combined after all other nodes are combined. Second, we assume that the number of the values of all nodes is the same in the algorithm. If the numbers of values of the nodes in a belief network are different, we can consider the product of the number of values of all nodes related in each step instead of the number of nodes. Third, a caching strategy can be used in the algorithm. A caching table is generated before any query. Before

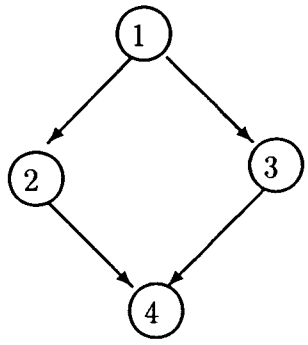


Figure 4. A simple belief network.

combining any two factors, we check the caching table to see if there is a cached result for the combination. If there is a cached result, we can use the cached result at a cost of 0 instead of doing the real probability computation. If there is no such cached result, then the real computation will be carried out. This caching strategy will save some computation time.

The heuristic strategy in the algorithm can be explained as follows. In step 2, $(x \cup y)$ shows the number of multiplications needed for combining the pair x and y. The number of multiplications is $2^{|x \cup y|}$. The elements in the set B are the candidates for the next combination. We don't consider pairs consisting of two unrelated marginal nodes if they don't have common children, since a combination of the two marginal nodes will usually increase dimensionality. In step 3, we choose the pairs which have the lowest result dimensionality as candidates since the best result of the current combination may need less multiplications than those of the other combinations for subsequent combinations. The effect of summation is considered here; it always decreases the dimensionality of the result. If more than one candidate is generated here, we choose the maximum $(|x| + |y|)$ in step 4 as a criterion because this choice maximizes the number of variables being summed over. Usually, it is better to sum over variables as early as possible. Steps 4 and 5 are just preparations for the next loop.

The time complexity of the algorithm is dominated by the number of nodes related to the current query. Step 1 is linear in the number of nodes. In step 2, there are $n(n-1)/2$ pairs to be computed for the set B at the first loop, and $(n-k)$ new pairs are added in the set at the end of the kth loop. There is a total of $(n(n-1)/2) + \sum_k(n-k) = (n^2 - 3n + 3)$ pairs to be computed. For each pair, the union operation is $O(m)$, here m is the maximum size of x; and sum(u) can be computed at the same time as computing $(x \cup y)$. So the time complexity in step 2 is $O(n^3)$ at most. The time cost of step 3 is linear in the number of pairs left in the set B and set C respectively; it is at most $O(n^2)$ including the $(n-1)$ loops needed for the step. The modification of the factor set in step 4 has linear cost in the number of factors, which is at most n. The deletion of elements from the set B in step 5 is linear in the number elements in the set. The time complexity is $O(n^2)$ in step 4 and $O(n^3)$ in step 5 including $(n-1)$ loops for the algorithm. Therefore, the time complexity of the algorithm is $O(n^3)$ in the number of nodes.

4.3.2 Experimental tests

The time complexity of some exact probabilistic inference algorithms, conditioning, clustering, reduction and SPI, has been analyzed, and their efficiency has been experimentally tested [Li90] using the IDEAL system [SB89] for conditioning, clustering and reduction algorithms and the implementation of SPI [D'A89]. Since SPI had better performance in that study than the others, we experimentally compare only set-factoring with SPI in this section.

Three sets of test cases were generated for time complexity experiments. We used J. Suermondt's random network generator to generate all test cases. This generator starts with a fully connected belief network of size n, and removes arcs selected at random until the number of the remaining arcs is equal to a selected value. In each test case, we randomly[8], (ranging from 1 to the number of nodes in

[8]Unless noted otherwise, all random selections are from uniform distributions over the indicated range.

the belief network), determined the number of observations to be inserted in that test case; then we randomly chose each observation from all unobserved variables in the belief network and finally we chose at random a set of variables as queries from remaining variables after each observation. The number of multiplications needed for each test case was recorded.

The first set of test cases is randomly generated with from 1.0 to 3.0 arcs per node and 8 to 13 nodes. The reason for choosing a set of small belief networks for testing is because we want to compare the results of set-factoring with those of an optimal algorithm, that is limited to small belief networks because of time complexity.[9] Table 1 shows the characteristics of the 10 test cases and the computational results of different algorithms measured in the number of multiplications. The data collected in this table are:

- **net**, the index of test cases;
- **node**, the number of nodes in each belief network;
- **arc/n**, the average arcs per node;
- **obs**, the number of observations inserted in the belief network;
- **qry**, the number of queries;
- **G.SPI**, the test results of the generalized SPI [SDD90];
- **set-f**, the test results of the set-factoring algorithm;
- **opt-alg**, the test results of an optimal factoring algorithm.

From the table we see that set-factoring has a better factoring result than the generalized SPI but is not optimal in two test cases.

The second set of test cases is tree-structured belief networks. They are randomly generated with from 10 to 30 nodes. Table 2 shows the 10 belief networks

[9]The optimal algorithm is a dynamic programming algorithm with exponential cost.

net	node	arc/n	obs	qry	G.SPI	set-f	opt-alg
1	12	2	3	7	287	52	52
2	11	2.5	3	7	328	196	196
3	9	2.5	4	12	301	252	252
4	11	2	4	4	58	26	26
5	9	2.2	1	3	140	102	102
6	8	2.6	2	4	200	194	186
7	13	1	3	7	109	38	38
8	13	2.5	3	8	2760	1818	1716
9	13	2.4	3	8	144	94	94
10	10	1.7	3	7	237	174	174

Table 1. Ten small test cases and the test results by algorithms: the generalized SPI, the set-factoring and the optimal algorithm.

and the test results. The columns 2 to 4 show the number of nodes, the number of observations and the number of queries for each test case. The columns 5 to 7 show the test results for each algorithm as in Table 1. From the table we see that set-factoring has an optimal result for each tree structured belief network. The generalized SPI did not get optimal results for some test cases.

The third set of test cases is that used in testing SPI and Generalized SPI [D'A89, D'A90, SDD90]. They are randomly generated from 1.0 to 5.0 arcs per node and 10 to 30 nodes. In Table 3, **n** shows the number of nodes and **arc** shows the number of arcs in each belief network; the columns **obs** and **qry** give the number of observations and total queries in each test case respectively; and the rest of the columns show the number of multiplications for each test case. A new version of SPI is used for comparison. **SPI-cache** and **set-f-cache** show the results with

net	node	obs	qry	G.SPI	set-f	opt-alg
1	23	6	68	728	646	646
2	19	19	89	1881	630	630
3	28	1	4	36	36	36
4	22	16	104	2959	1246	1246
5	17	7	34	809	404	404
6	12	9	27	335	148	148
7	24	17	128	1469	68	68
8	25	1	10	222	178	178
9	24	5	58	1478	1010	1010
10	22	5	46	1427	642	642

Table 2. Tree structured test cases and test results by algorithms: the generalized SPI, the set-factoring and the optimal algorithm.

intermediate result caching for both algorithms.[10]

From the above experimental results we see that the factoring strategy of set-factoring has better factoring results than those of SPI in every case, particularly when a belief network is large. The number of multiplications in set-factoring is about half of those in SPI on average. Set-factoring is more consistent with respect to tasks and different kinds belief networks. As shown in table 3, set-factoring is better than SPI with caching for a large belief network, taking network 16 as an example. Since dimension in a factor will become large after some combinations, any bad combination order will cause many more multiplications than a good combination does.

The time complexity of factoring for set-factoring and the time complexity of symbolic reasoning for SPI are only slightly different. In set-factoring, the time

[10]* denotes that corresponding algorithm is too slow to run the test case.

net	n	arc	obs	qry	SPI	set-f	SPI-cache	set-f-cache
1	23	28	10	13	164	98	140	60
2	13	62	7	6	832	718	368	310
3	13	61	10	4	62	44	32	28
4	18	85	10	8	624	558	422	418
5	16	54	8	9	2,370	1,512	866	898
6	17	34	8	9	2,616	890	1,176	502
7	23	60	10	12	37,514	5,272	10,078	2,978
8	10	15	5	5	286	182	222	92
9	27	35	13	14	1,122	644	800	244
10	12	26	5	7	780	386	452	194
11	23	87	10	12	183,296	73,804	65,216	26,540
12	11	36	5	6	1,896	1,126	668	598
13	14	15	7	6	454	228	264	92
14	16	40	8	8	8,416	3,112	2,204	1,940
15	19	76	9	10	81,696	23,590	13,380	10,462
16	29	131	1	28	*	6,569,756	16,146,192	3,196,900
17	29	90	14	14	1,489,040	143,334	254,292	73,146
18	16	35	9	6	2,480	898	816	450
19	15	53	7	8	15,986	4,168	3,068	1,896
20	26	101	13	13	717,552	124,734	113,248	63,834
21	28	34	14	13	2,052	847	1,384	330

Table 3. The experimental results of 21 test cases between SPI and set-factoring.

complexity is $O(n^3)$ in the number of nodes concerned in the current query at most; in SPI the time complexity is $O(n^3)$ at most in the number of nodes of the belief network. There should be no big difference. The time cost for symbolic reasoning in both algorithms is trivial compared to probability computation.

4.3.3 Discussion

While these results are preliminary, they seem a strong indication that the set-factoring algorithm is able to find better factoring for many problems, particularly in finding optimal factoring for all tree test cases. Also the set-factoring algorithm can be used as a suitable analytical tool for evaluating other probabilistic inference algorithms. The most important conclusion from the experimental results is that OFP is a useful way of efficiently solving probabilistic inference problems in a belief network. From the OFP point of view, not only can we get a better algorithm than those previously developed, but also the algorithm is easy to understand and implement.

The main idea behind the set-factoring algorithm is, at each step, to find a pair with the best combination result. We tried the strategy of finding the pair with minimum multiplication as a candidate for combination; the results are not as good as those obtained by set-factoring. The set-factoring algorithm only considers information locally for choosing each pair, so it can be implemented efficiently. It is this characteristic that prevents the algorithm from guaranteeing an optimal result for some multiply-connected belief networks, because optimal results are related to all nodes concerned. It also tells us why the algorithm is good in tree structured belief networks; the factoring information for the tree is locally determined. Due to the locality of its heuristic strategy, set-factoring can work as a local factoring strategy in other probabilistic inference algorithms.

Since the last several combinations in factoring usually have large dimensionality, combinations of the last few factors are critical in getting nearly optimal results for some belief networks. Considering this, we combined the set-factoring

and the optimal algorithm together to get a new algorithm in which we used set-factoring to generate a partial result first and then used the optimal algorithm to complete the last several combinations. Since the optimal algorithm can run efficiently for about 8 factors, the combined algorithm should run efficiently as well. The results of the combined algorithm are better than the set-factoring algorithm, particularly for large belief networks[11]. This led us to think of another factoring strategy of using the optimal algorithm. That is, if a belief network can be divided into several connected parts, we might use the optimal algorithm within each part and then among all parts. We have not tested this idea yet.

The test result of the network 3 in table 3 for set-factoring (without caching) is optimal for each query, but both algorithms with caching give better results for the same queries. This indicates that a best probabilistic inference algorithm may not only depend on an optimal factoring strategy, but also depend on a good caching method for some tasks and some belief networks. There is a trade off between using a good factoring strategy and using an effective caching method in an inference algorithm, since a good factoring strategy, flexible across many belief networks and tasks, may be hard to combine with any caching method.

In chapter 6 we will study the problem of parallelizing probabilistic inference in belief networks. Set-factoring has shown good factoring results for parallelizing probabilistic inference.

[11]Take the network 16 in table 3 as an example, the number of multiplications needed by the combined algorithm is about 75% of those by set-factoring.

Chapter 5

Finding l Most Probable Explanations in Belief Networks

5.1 Motivation

Finding the *Most Probable Explanation*(MPE) [Pea88] of a set of evidence in a Bayesian (or belief) network is the task of the identification of an instantiation or a *composite hypothesis* of all nodes except the observed nodes in the belief network; such that the instantiation has the largest posterior probability over all such instantiations. Since the MPE provides the most probable state of a system, this technique can be applied to system analysis and diagnosis. Finding the l most probable explanations of a given evidence is the identification of the l instantiations with the l largest probabilities.

There have been several research efforts for finding MPE in recent years and several methods have been proposed for solving the problem. These methods can roughly be classified into two different groups. One group studying the problem of finding the best explanation for text [HSME88, CS90, SC90a] considers MPE as the problem of minimal-cost-proofs. In finding the minimal-cost-proofs, belief networks are converted to Weighted Boolean Function Directed Acyclic Graphs (WBFDAG) [SC90a], and then best-first search techniques are applied to find the MPE in the WBFDAGs. Although this approach has a linear run time in the size of

the converted graph, the number of the nodes in the converted graph is exponential in the size of the original belief network. In order to improve the performance of the minimal-cost-proof technique, a new technique, which translates the minimal-cost-proof problems into 0-1 programming problems, has been presented; and the 0-1 programming problems can be solved by using the simplex method combined with branch and bound techniques [San89, San91, CJ92]. Although the new technique outperformed the best-first search technique, there are some limitations to using it: such as that the original belief networks should be small and their structures must be close to and-or dags. The second group of methods directly evaluate belief networks for finding the MPE but restrict the type of belief networks to singly-connected belief networks [Pea88, Sy92a, Sy92b] or to a particular type of belief networks: such as BN2O [HD90] or bipartite graphs [Wu90]. Arbitrary multiply-connected belief networks must be first converted to singly-connected networks and then can be solved by these methods. The algorithm developed by J. Pearl [Pea88] presents a message passing technique for finding the two most probable explanations; but this technique is limited to only finding two explanations [Nea90] and can not be applied to multiply-connected belief networks. Based on the message passing technique, another algorithm [Sy92a, Sy92b] has been developed for finding the l most probable explanations. This algorithm has some advantages over the previous one: its message passing strategy is unidirectional; the message unit is a vector not a value; and the intermediate results are cached for successive computation. However, this algorithm is limited to singly-connected belief networks; also, methods for choosing a "sink node" and determining message passing paths are not discussed in the paper.

In this chapter, we will discuss the essence of the MPE problem and propose a framework, optimal factoring, for handling this problem. Using this framework we will develop an optimal algorithm for finding the MPE in a singly-connected belief network and an efficient algorithm for finding the MPE in a multiply-connected belief network. We will present methods for finding l MPEs directly for either

singly-connected belief networks or multiply-connected belief networks. We will also describe ways of finding the l MPEs for an arbitrary subset of nodes in a belief network.

The rest of the chapter is organized as follows. Section 2 discusses the essence of MPE problem. Section 3 presents an algorithm for finding the MPE in a belief network. Section 4 proposes a framework, optimal factoring, for solving the MPE problem. Section 5 presents an optimal factoring algorithm for singly-connected belief networks, and section 6 presents an efficient factoring algorithm for multiply-connected belief networks. Section 7 presents a linear time algorithm for finding next MPE after finding the first MPE. Section 8 discusses the issue of finding the MPE for a subset of variables in a belief network. Section 9 introduces some of the related work. And finally, section 10 summarizes the research.

5.2 The essence of the MPE problem

Finding the l Most Probable Explanations (MPEs) of a given evidence, S_e, in a belief network can be formulated as identifying and ordering l instantiations (or l composite hypotheses), H_is, for which the posterior probabilities are the largest, i.e., $Pr(H_1|S_e) \geq ... \geq p(H_l|S_e)$.

The computational cost for finding the MPE in a belief network depends on the structure of the belief network. Finding the MPE may be as hard as the problem of querying the marginal probability of a node in some cases.

The idea of solving the MPE problem is similar to that of solving the problem of querying the marginal probability of a node in a belief network. In computing the marginal probability of a node, we could compute the full joint probability of the belief network and then sum over all non-queried nodes. This method can be improved by performing the summation for some variables earlier in the computation. Similarly, the MPE can be found by computing the full joint probability of

the belief network and choosing the largest element. This computation can also be improved by performing comparison operations for instantiations of some variables early in the computation.

The problem of finding the l MPEs is related to the problem of finding the full joint probability of a belief network. If we know the full joint probability of a belief network, we can easily obtain the l MPEs by sorting the joint probability table in descending order and choosing the first l instantiations. However, this method would be quite inefficient. The time complexity for computing a full joint probability is exponential with respect to the number of nodes in the belief network. There is no significant difference between any approaches in finding l MPEs in a fully connected belief network because the cost of computing the full joint probability in the belief network is linear with respect to the input size of the problem, which in turn is exponential in the number of nodes. For non-fully-connected belief networks, some approaches may be more efficient than the approach of computing the full joint probability. The approach proposed for singly-connected belief networks in [Sy92a] is an example. The MPE can be calculated by the right hand side of the formula (1.1) with an instantiation of the n variables. Since the right hand side formula is determined by the structure of a belief network, with its probability distributions, we should find a way of choosing a proper instantiation in the formula.

Finding the MPE may be considered as a search problem. If we create a tree from the n variables in a belief network in which a node represents a variable and arcs from a variable indicates all possible values of that variable, then a path from the root node to any leaf node gives one possible instantiation of all variables in the belief network. Then, the MPE could be found by searching all paths of the tree. There are 2^{n-1} different paths in the tree (if each node has two values[12]) and search complexity is 2^n in the worst case. Therefore, considering the problem of finding the MPE as a search problem is not an efficient method.

[12]Since the basic operation in a search algorithm is comparison, it is more appropriate to consider the number of comparisons for describing the cost of search in solving the problem.

We think that the costliest computation in finding the MPE is not in search or comparison but in probability computation. In order to analyze the computational complexity of the MPE problem, we can consider the whole computation in finding the MPE as consisting of two parts. One is *comparison*, which compares a set of values and chooses the largest one. The other is probability computation, which multiplies some distributions together to form a new distribution. If we quantitatively measure the cost of finding the MPE as the number of comparisons for comparison and the number of multiplications for probability computation, the number of multiplications is usually higher than that of comparisons. Higher cost in probability computation can be explained as follows. Assume we have a result distribution of a conformal product with n variables v_1, v_2,...,v_n and their domain sizes a_1,...,a_n respectively. The number of multiplications of the conformal product is $a_1a_2...a_n$. If we want to get the largest instantiations of variable v_1 with arbitrary instantiations of the other variables, the number of comparisons needed is $(a_1-1)a_2...a_n$. It is obvious that the number of comparisons is smaller than that of the multiplications. Similarly, if we want to get the largest instantiations of variable v_2 after instantiating v_1, the number of comparisons is $(a_2-1)a_3...a_n$. We can find the largest instantiations for each variable until a_n. It is easy to see that the total number of comparisons for finding the largest instantiation of all variables above is $(a_1a_2...a_n)-1$ which is less than the number of multiplications. Efficiently finding the MPE depends on comparing instantiations of some variables, eliminating them from a distribution at a right time, and reducing probability computation as much as possible.

5.3 An algorithm for finding the MPE

We will discuss the comparison and probability computation separately so that we can understand how to reduce the two different costs in finding the MPE. We begin with a definition.

Definition *A distribution over a set of variables* Π *is Reduced with respect to one of its variables* α *by selecting, for each instantiation of the remaining variables* $\Pi \setminus \alpha$, *the largest of the corresponding entries (ranging over instantiations of* α*) in the distribution.*

Lemma 5.1 tells us when we can reduce a distribution during an MPE computation.

Lemma 5.1 *Given a belief network, if a node* x *has no descendants, the conditional distribution (or marginal distribution if the node has no antecedents) of this node can be reduced with respect to* x *for purposes of finding the MPE; the ratio of the reduced distribution to the original distribution is* $1/|x|$.

Proof: Assume that the node x has domain values $X_1, ..., X_k$, and parent variables $y_1, ..., y_l$. Since the node x has no descendants, there is only one possible instantiation of x that qualifies for finding the MPE in any instantiation of variables $y_1, ..., y_l$. That is, we only need

$$max(p(x = X_1|y_1 = Y_1, ..., y_l = Y_l), ..., p(x = X_k|y_1 = Y_1, ..., y_l = Y_l)$$

for each instantiation of y_is. Therefore, we only need to keep the result of the above *max* operation for each instantiation of the y_is . The ratio of the size of the reduced distribution to the original distribution is $1/|x|$. ∎

Lemma 5.2 tells us when we can reduce an intermediate result distribution (a distribution resulting from a conformal product operation).

Lemma 5.2 *A result distribution of a conformal product can be reduced for finding the MPE for all variables in the distribution which don't appear in any other remaining distribution. The ratio of the size of the reduced distribution to the size of the result distribution is* $1/\Pi_{r \epsilon R}|r|$, *where* R *is the set of variables over which the reduction is performed.*

Proof: The proof is very similar to the proof of lemma 5.1. If a variable only appears in one distribution, it must be a conditioned variable in that distribution, and its role in finding the MPE is the same as that of a node which has no descendents in the original belief network. Therefore, it can be treated similarly. We can reduce the distribution one variable at a time as discussed in the lemma 5.1, and the ratio of the reduced distribution to the original distribution is one over the product of the domain size of those variables which are not appearing in any other distribution. ∎

It is important to understand the meanings of the two lemmas although they are simple. Given a belief network, we can not always determine an instantiation for the MPE by just glancing at the values in all distributions. Because one instantiation of a variable with the largest value in one distribution may not have the largest value in another distribution, the product of all values from all distributions may not be the largest one. The difficulty of the MPE problem lies in that one variable could blend in more than one distribution; and its largest instantiation could not be determined by part of these distributions. The lemmas tell us when an instantiation of a variable can be determined and reduced to a small portion of the original distribution containing the variable. Also, the chosen instantiations of a variable are guaranteed to contain an instantiation which contributes to the MPE.

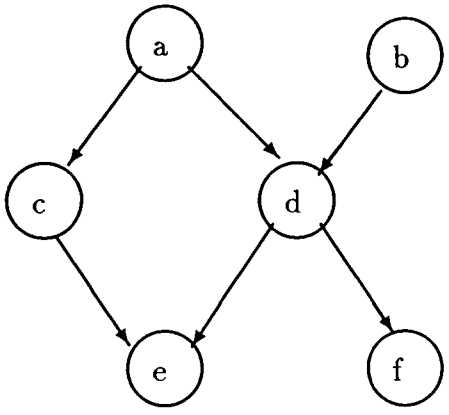
Given the two lemmas, we have an algorithm for finding the MPE of a belief network:

1. Apply lemma 5.1 to any distributions if applicable;

2. Create a factoring for these distributions;

3. Combine these distributions according to the factoring and apply lemma 5.2 to the result of each conformal product as applicable [13].

[13]Steps two and three can be mixed together by finding a partial factoring for some distributions and combining them.



p(a): p(a=1)=0.2
p(b): p(b=1)=0.3
p(c|a): p(c=1|a=1)=0.8 p(c=1|a=0)=0.3
p(d|a,b): p(d=1|a=1,b=1)=0.7
p(d=1|a=1,b=0)=0.5
p(d=1|a=0,b=1)=0.5
p(d=1|a=0,b=0)=0.2
p(e|c,d): p(e=1|c=1,d=1)=0.5
p(e=1|c=1,d=0)=0.8
p(e=1|c=0,d=1)=0.6
p(e=1|c=0,d=0)=0.3
p(f|d): p(f=1|d=1)=0.2 p(f=1|*d*=0)=0.7

Figure 5. A simple belief network.

The largest instantiated value in the final result is the MPE.

We have discussed the comparison portion of finding the MPE in belief networks; we will discuss the probability computation portion now. The probability computation refers to conformal products. The two portions of computations for finding the MPE can be illustrated by using the belief network in figure 5.

Given the belief network in figure 4, we want to compute its MPE. There are six distributions in the belief network. If we use $D(x, y)$ to denote a distribution with variables x and y in it[14] and $d(x = 1, y = 1)$ to denote one value in $D(x, y)$, [15] then the six original distributions are: $D(a)$, $D(b)$, $D(a, c)$, $D(a, b, d)$, $D(c, d, e)$ and $D(d, f)$. We can apply lemma 5.1 to the distributions related to nodes e ($D(c, d, e)$ and f ($D(d, f)$). The result distribution with instantiations for f is $D(d)$:

$d(d = 0) = 0.7$ *with* $f = 1$ $\qquad$ $d(d = 1) = 0.8$ *with* $f = 1$

and $D(c, d, e)$ becomes $D(c, d)$:

$d(c = 1, d = 1) = .5$ *with* $e = 1$ $\qquad$ $d(c = 1, d = 0) = .8$ *with* $e = 1$

[14]Which variable is a conditioned variable, which is a conditioning variable, and the order of appearance of these variables, is not important.

[15]We will use the denotation with the same meanings here in the rest of the paper.

$d(c = 0, d = 1) = .6$ *with* $e = 1$ $\quad$ $d(c = 0, d = 0) = .7$ *with* $e = 0$

The other four distributions are the same as those in Figure 5. We can combine these in any order, applying lemma 5.2 to each intermediate result. Suppose the combination order is:

$$(((D(a) * D(a, c)) * (D(b) * D(a, b, d))) * (D(c, d) * D(d))).$$

We can first combine $D(a)$ with $D(a, c)$, obtaining a new $D(a, c)$:

$d(a = 1, c = 1) = .16$ $\quad$ $d(a = 1, c = 0) = .24$

$d(a = 0, c = 1) = .04$ $\quad$ $d(a = 0, c = 0) = .56$

Lemma 5.2 can not be applied to the result. Combine $D(b)$ with $D(a, b, d)$, the result $D(a, b, d)$ is:

$d(a = 1, b = 1, d = 1) = .21$ $\quad$ $d(a = 1, b = 1, d = 0) = .09$

$d(a = 1, b = 0, d = 1) = .35$ $\quad$ $d(a = 1, b = 0, d = 0) = .35$

$d(a = 0, b = 1, d = 1) = .15$ $\quad$ $d(a = 0, b = 1, d = 0) = .15$

$d(a = 0, b = 1, d = 0) = .14$ $\quad$ $d(a = 0, b = 0, d = 0) = .56$

Apply lemma 5.2 to $D(a, b, d)$ and instantiate variable b, the result is $D(a, d)$:

$d(a = 1, d = 1) = .35$ *with* $b = 0$ $\quad$ $d(a = 1, d = 0) = .35$ *with* $b = 0$

$d(a = 0, d = 1) = .15$ *with* $b = 1$ $\quad$ $d(a = 0, d = 0) = .56$ *with* $b = 0$.

Combine $D(c, d)$ with $D(d)$, the result is $D(c, d)$:

$d(c = 1, d = 1) = .4$ *with* $e = 1$ $f = 0$

$d(c = 1, d = 0) = .56$ *with* $e = 1$ $f = 1$

$d(c = 0, d = 1) = .48$ *with* $e = 1$ $f = 0$

$d(c = 0, d = 0) = .49$ *with* $e = 0$ $f = 1$

and lemma 5.2 can't be applied. Then, combining $D(a, c)$ with $D(a, d)$ results in $D(a, c, d)$:

$d(a = 1, c = 1, d = 1) = .056$ *with* $b = 0$

$d(a = 0, c = 1, d = 1) = .0360$ *with* $b = 1$

$d(a = 1, c = 1, d = 0) = .056$ *with* $b = 0$

$d(a = 0, c = 1, d = 0) = .1344$ *with* $b = 0$

$d(a = 1, c = 0, d = 1) = .014$ *with* $b = 0$

$d(a = 0, c = 0, d = 1) = .0840$ *with* $b = 1$

$d(a = 1, c = 0, d = 0) = .014$ *with* $b = 0$

$d(a = 0, c = 0, d = 0) = .3136$ *with* $b = 0$

Applying lemma 5.2 to the result, results in $D(c, d)$:

$d(c = 1, d = 1) = .056$ *with* $a = 1$ $b = 0$

$d(c = 1, d = 0) = .1344$ *with* $a = 0$ $b = 0$

$d(c = 0, d = 1) = .084$ *with* $a = 0$ $b = 1$

$d(c = 0, d = 0) = .3136$ *with* $a = 0$ $b = 0$

Finally, combine the $D(c, d)$ with instantiated a and b with $D(c, d)$ with instantiated e and f, a new $D(c, d)$ is:

$d(c = 1, d = 1) = .022400$ *with* $a = 1$ $b = 0$ $e = 1$ $f = 0$

$d(c = 1, d = 0) = .075264$ *with* $a = 0$ $b = 0$ $e = 1$ $f = 1$

$d(c = 0, d = 1) = .040320$ *with* $a = 0$ $b = 1$ $e = 1$ $f = 0$

$d(c = 0, d = 0) = .153664$ *with* $a = 0$ $b = 0$ $e = 0$ $f = 1$

Choose one item with the largest value in $D(c, d)$, we obtain the MPE. The MPE is $p(a = 0, b = 0, c = 0, d = 0, e = 0, f = 1)$.

We can see, from the above example, that the number of multiplications is comparable to the number of comparisons in finding the MPE. We know that the number of multiplications needed in a conformal product is exponential in the dimensionality of the distribution of the conformal product. From lemma 5.2 we know that the number of comparisons needed for choosing a proper distribution is exponential in the dimensionality of the result distribution. If we could reduce the dimensionality of a distribution, both multiplications and comparisons needed for the distribution would be reduced in exponential proportion. This indicates that reducing the dimensionality of a conformal product results directly in an efficient algorithm for finding the MPE. That is, step 2 of the above algorithm is crucial. Therefore, efficiently finding the MPE can be considered as the problem of finding

an optimal factoring in step 2 of the algorithm. In the following three sections we will discuss the factoring problem.

5.4 Optimal Factoring Problem for finding the MPE

In this section and the following two sections, we will discuss the problem of creating a factoring for finding the MPE (the second step of the algorithm for finding the MPE). We have discussed that the problem of finding the MPE is related to the problem of computing the full joint probability in a belief network. We believe that efficiently finding the full joint probability of a belief network is an optimal factoring problem; so is the probability computation in finding the MPE. We have defined an optimal factoring problem in chapter 3 for probabilistic inference in belief networks. Similarly, we will define another optimal factoring problem here and then discuss how the optimal factoring problem helps to find the MPE.

Definition [MPE-FP] *Given*

1. *a set of m variables V,*
2. *a set of n subsets of V:* $S = \{S_{\{1\}}, S_{\{2\}}, ..., S_{\{n\}}\}$, *and*
3. $Q \subseteq V$ *is a set of target variables;*

define operations:

1. *combination of two subsets* S_I *and* S_J*:*

$$S_{I \cup J} = S_I \cup S_J - \{v : v \notin S_K \text{ for } K \cap I = \phi, \\ K \cap J = \phi, \text{ and } v \notin Q\},$$

$$I, J \subseteq \{1, 2, ..., n\},\ I \cap J = \phi;$$

2. *the first cost function of combining the two subsets:*

$$\mu(S_{\{i\}}) = 0 \text{ for } 1 \leq i \leq n, \text{ and}$$

$$\mu(S_{I \cup J}) = \mu(S_I) + \mu(S_J) + 2^{|S_I \cup S_J|};$$

3. the second cost function of combining the two subsets:

$$\eta(S_{\{i\}}) = 2^{|S_{\{i\}}|-1} \; if \; \exists \, w, \; w \in S_{\{i\}} \wedge w \notin S_{\{j\}},$$

$$otherwise \;\; \eta(S_{\{i\}}) = 0, \; for \; 1 \leq i, \; j \leq n \;\; \wedge i \neq j \; and$$

$$\eta(S_{I \cup J}) = \eta(S_I) + \eta(S_J) + \sum_{i=1}^{|v|} 2^{|S_I \cup S_J|-i}, \; if \; \{v\} \neq \phi,$$

$$otherwise \;\; \eta(S_{I \cup J}) = \eta(S_I) + \eta(S_J).$$

A factoring problem is to find a way of combining the n subsets in S to lower the combination cost.

The definition here is the FP definition in chapter 3, plus an additional cost function, η, for comparisons. We keep the two costs separate so that we can directly cite results from chapter 3 where relevant. In above definitions, Q is a set of target variables to be retained in the result. The set $\{v\}$ in the formula $S_{I \cup J}$ contains a set of variables which do not appear in the remaining subsets of S after combination of S_I with S_J or in Q. $\mu_(S_{I \cup J})$ and $\eta_(S_{I \cup J})$ are the two costs of combining all set S_i $(i \in I, J)$ in a given factoring order. They are determined by the dimensionality or the size of sets to be combined and affected by the size of $\{v\}$ in previous combinations. If the domain size of each variable is not limited to 2, the value $2^{|S_I \cup S_J|}$ in the above formula should be replaced with the product of the domain size of the variables in $S_I \cup S_J$.

$\mu(S_I)$ and $\eta(S_I)$ are not unique if $|I| > 2$ in the above definition. In general, they depend on how we combine the subsets. We indicate these alternative combinations by subscripting μ and η. For example, $\mu_\alpha(S_I) = \mu$ shows the cost of combining result of S_I with respect to a specific tree structured combination of I, labeled α. We call this combination a factoring. We did not give a total cost for μ and η in the definition, because they may not be measured in the same units. We might do so if they could be scaled in one unit. Since both μ and η are determined by a factoring and if μ reflects the major combination cost, we can define an optimal factoring based on the μ cost of combinations.

Considering that if an individual operation cost for μ is larger than an individual operation cost for η, and the power proportion for μ is always bigger than that for η, and the number of μ operations is never less than the number of η operations, we can simplify our consideration for optimal factoring only in the μ cost in the FP definition. Therefore, we can give a formal definition for optimal factoring.

Definition [MPE-OFP] *An* **optimal factoring** *problem is to find a factoring* α *such that* $\mu_\alpha(S_{\{1,2,...,n\}})$ *is minimal.*

All possible factorings of n variables are equivalent to the results of permuting the n variables and then putting parentheses in all legal ways in the permutation to form all $S_{\{1,2,...,n\}}$. The optimal factoring may not be unique.

Given the definition of MPE-OFP, we can map the problem of finding the MPE in a belief network to an instance of MPE-OFP. The mapping is the same as what we discussed in chapter 3. After the mapping, μ gives the number of multiplications needed for the computation, and $\eta_(S_{I\cup J})$ is the number of comparisons needed for reducing the result distribution ($S_{I\cup J}$), α gives a particular combination order of these n distributions. The cost of each comparison operation in η is usually smaller than the cost of floating point multiplication in μ; the number of μ operations is greater than or equal to the number of η operations; and the maximum dimensionality of μ operations is greater than that of η operations. Therefore, we can ignore the cost of η in time complexity analysis for solving the MPE problem. This is analogous to ignoring addition cost in probabilistic inference in OFP. The purpose of mapping the problem of finding the MPE to OFP is to apply some techniques used for solving combinatorial optimization problems to OFP and apply some results in OFP to the MPE problem. It should be clear that determining the lowest cost way of finding the MPE in a belief network is an optimal factoring problem.

It is interesting to know the difference between querying posterior probability of some variables and finding the MPE in the OFP framework. First, there are target variables or queried variables in posterior probability computation; while

there are no target variables in finding MPE. So, the computation for a posterior probability computation is query relevant, namely only the nodes relevant to the query are in the computation; but, finding the MPE is related to the whole belief network. This difference expressed in the FP definition is whether Q is empty or not. Second, marginalization in posterior probability computation is replaced by *max* operations in finding the MPE, but the number of operations in both cases are exactly the same. Third, distributions for variables which have no direct descendants should be reduced at the beginning of finding the MPE; while some queried variables should not be summed over in posterior probability computation. From these differences we find that the procedure for posterior probability computation should be very similar to the procedure of finding the MPE; and the computation time for finding the MPE should be less than that for querying a posterior probability when the same distributions are involved in both computations. We also note that the maximum dimensionality provides a way of measuring the time complexity in both computations.

The MPE-OFP definition provides a way of quantitatively measuring the efficiency of an algorithm and also provides a way of considering how to find efficient algorithms. In next section, we will show how the factoring framework provides a way of generating an optimal factoring for finding the MPE in a singly-connected belief network.

5.5 Optimal factoring in singly-connected belief networks

From the discussion in section 5.2, we know that if we can find an optimal factoring for a belief network we can find the MPE efficiently. Some results for the factoring problem in chapter 3 can be used for finding the MPE. In particular, by using the lemma 3.2 to lemma 3.5 in that chapter, we can show that there exists an optimal factoring algorithm for a singly-connected belief network with linear time cost.

Theorem 5.1 *There exists a linear time algorithm to generate an optimal factoring for finding the MPE in a singly-connected belief network.*

Proof: We can construct an optimal factoring algorithm for a singly-connected belief network from the above lemmas. Given a singly-connected belief network, the factoring algorithm is as follows.

Choose any group in the belief network and put the group in a stack. Repeat the following computation until the stack is empty:

1. Consider the group at the top of the stack (called current group). If the group has only one node which is also part of another group then
   (a) Apply lemma 5.1 to the group head of the current group if the lemma is applicable.
   (b) If there exist some distributions of dimension 1 which share a variable and that variable is the current group head, then combine those distributions together. (These distributions may come from previous computation or occur as a result of evidence.)
   (c) Apply lemma 3.2 or lemma 3.3, whichever applicable, and lemma 5.2 (if applicable) to the current group. The result distribution will be used for the following computation.
   (d) Pop the current group off the stack and push its connected group(s), if they are not in the stack, on the stack in any order.
2. Otherwise, push all groups connected with the top group of the stack into the stack in any order.

The MPE can be obtained by choosing the largest value of the final result distribution with its associated instantiation.

The steps in the factoring algorithm can be explained as follows. Let us first consider the case where there is only one node in the current group connected to other groups. The first step (a) in the above algorithm is to reduce the distributions of the nodes which have no child; that is, to find instantiations of those nodes so that their instantiation will lead to possible MPE and ignore the other instantiations. The work in the second step (b) is just as mentioned in that step. The reason for the step is lemma 3.4. Take figure 3(c) as an example: after the first step in the algorithm for nodes 1 to n, there are n distributions with the only variable y in them; we should combine them together to find the maximum value of y. The third step (c) is the main part of the algorithm, which combines a group and reduces all variables in the group except one which connects to other groups. It is important to notice that when a group has less than two nodes in connection with other groups, their relations satisfy the condition of either lemma 3.2 or lemma 3.3. When applying either of the lemmas, the node connecting other groups is the node in Q in the lemmas. After applying either of the two lemmas and lemma 5.2, all nodes except the nodes in Q are instantiated. The final step (d) continues these computations by pushing un-processed groups onto the stack. The method of popping and pushing groups in the stack ensures that each group will be processed exactly once. If the current group has more than one node connecting some other groups, we can't apply lemmas 3.2 or 3.3 because these nodes appear in more than two distributions and can not be instantiated according to the lemma 5.2. We keep the current group and compute its connected groups first. If these groups can be computed, the previously kept group can then be computed. If these groups can't be computed for the same reason, we keep these groups and consider their connected groups, and so on. This recursive process will be terminated because any group at the edge of a singly-connected belief network satisfies either lemma 3.2 or 3.3. We will give a simple example to illustrate the algorithm after a discussion of optimality.

The optimality of the algorithm is easy to understand. The lemma 3.5 tells us that cost of combining two nodes in two groups is always higher than combining two

nodes in the same group. In the algorithm we complied with this rule. Furthermore, lemmas 3.2 and 3.3 choose optimal combinations within a group. Therefore, the algorithm is optimal.

The time complexity of the algorithm is linear in the number of nodes in a belief network. This can be seen by separating factoring steps from the conformal product and reduction operations. The factoring steps creating an evaluation tree for finding the MPE, the conformal products combine two distributions, and the reduction operation compares alternate instantiations. It is very clear that the first and the second steps, (a) and (b), in the algorithm take linear time in the number of nodes in a belief network. In the third step (c), each group is put into the group stack once and the nodes in the group are examined at most twice for their connectivity (we can mark each group in the stack so that any group needs to be checked once). The order of popping out groups gives us an evaluation tree. Therefore, the algorithm is linear in time with respect to the number of nodes in a belief network. ∎

The computational cost for finding the MPE under OFP framework is mainly the number of multiplications, which is determined by the dimensionality of each conformal product of the distributions in a belief network and dominated by the maximum dimensionality of the conformal products. The time complexity for finding the MPE is exponential with respect to the maximum dimensionality. In a singly-connected belief network, the maximum dimensionality is the size of the largest group under the OFP framework. If the maximum dimensionality of a factoring result is n, and the number of groups is k, then, the time complexity for finding the MPE is $O(k2^n)$.

Following is a simple example to illustrate the algorithm for a singly-connected belief network. The simple belief network is given in figure 6. In the example, we use G_c to represent a group and the subscript c to indicate the group head. $\{G_c, G_e\}$ represents a stack with two elements in it, and the top of the stack is G_c. The computation steps and intermediate results are listed as follows.

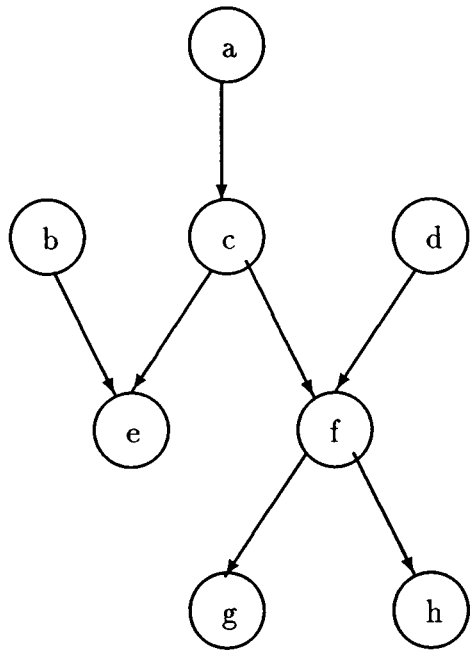


Figure 6. A singly-connected belief network.

1. Choose group G_f (randomly chosen) put in the stack: $\{G_f\}$.

   (a) Loop1:

   Since there is more than one node in G_f connected to other groups, all connected groups are put into the stack in any order.

   The stack is $\{G_e, G_c, G_g, G_h, G_f\}$.

   (b) Loop2:

   Since only node c in G_e is connected to some other groups, the four inner steps can be applied to this group. In step 1, $D(b, c, e)$ is changed to $D(b, c)$; the variable e is instantiated and eliminated by comparison; step 2 is not applicable to G_e; in step 3, lemma 3.2 is applied to the conformal product $D(b, c) * D(b)$. The result distribution is $D(c)$ and the variable b is instantiated and eliminated. Finally, after the fourth step, the stack is $\{G_c, G_g, G_h, G_f\}$.

(c) Loop3:
distributions $D(a,c)$, $D(a)$ and $D(c)$ are combined together.
The result distribution is $D(c)$, and the resulting stack is $\{G_g, G_h, G_f\}$.

(d) Loop4: the variable g is instantiated and the distribution $D(f,g)$ is reduced to $D(f)$. The stack is $\{G_h, G_f\}$.

(e) Loop5: the variable h is instantiated and the distribution $D(f,h)$ is reduced to $D(f)$. The stack is $\{G_f\}$.

(f) Loop6: step 1 is not applicable; in step 2, conformal product $D(f)*D(f)$ is carried out to form a new $D(f)$; in step 3, distributions $D(c,d,f)$, $D(c)$, $D(d)$ and $D(f)$ are combined together. The variables c and d are instantiated and eliminated. The result distribution is $D(f)$.

2. After the loop6, the stack is empty. The MPE is determined by the largest value of $D(f)$, which carries the instantiations for the other variables as determined in previous loops.

If we define an unary operator Φ_a for a probability distribution $p(c|a)$, i.e. $\Phi_a p(c|a)$, to indicate the operation of instantiating the variable a and eliminating the variable a from distribution $p(c|a)$, the operations above for finding the MPE can be represented as:

$$(\Phi_{c,d}(p(f|c,d)*((\Phi_b(\Phi_e p(e|b,c)*p(b))*\Phi_a(p(c|a)*p(a)))*p(d))))*(\Phi_g p(g|f)*\Phi_h p(h|f)).$$

We have shown that there exits an optimal factoring algorithm for finding the MPE for a singly-connected belief network. We will discuss how to find the MPE in multiply-connected belief networks in the next section.

5.6 Factoring in multiply-connected belief networks

Finding an optimal factoring for the MPE in multiply-connected belief networks is not as easy as in singly-connected belief networks. MPE-OFP is very similar to

OFP from their definitions. That is, MPE-OFP generally is a hard problem, and we don't expect to find an efficient optimal factoring algorithm for finding the MPE for an arbitrary belief network.

Since the problem of finding the MPE is very similar to the problem of posterior probability computation in the OFP framework, we may apply some results for posterior probability computation in chapter 3 to the problem here. For example, we can use the dynamic algorithm to find an optimal factoring for finding MPE with exponential time cost. Since the general OFP appears to be a hard problem, we have to turn our attention to finding a heuristic algorithm and find an algorithm as close to an optimal factoring as possible. We have developed a heuristic algorithm in chapter 4, which runs efficiently for posterior probability computation. We can use it with little modification for finding the MPE since the evaluation tree generated for finding the MPE is the same as the evaluation tree for posterior probability computation with the same distributions.

Applying the factoring algorithm in chapter 4 for finding the MPE of the belief network in figure 4, the factoring result is

$$\Phi_{2,3}(\Phi_4 p(4|2,3) * (\Phi_1(p(3|1) * ((p(2|1) * p(1)))))).$$

Like the case for a singly-connected belief network, the computational cost for finding the MPE in a multiply-connected belief network is determined mainly by the number of multiplications of each conformal product and dominated by the maximum dimensionality of these conformal products. The time complexity for finding the MPE is exponential with respect to the maximum dimensionality which depends on the quality of a factoring algorithm.

5.7 Finding l MPEs in belief networks

In this section, we will show that the algorithm presented in section 5.2 provides an efficient way for finding the l MPEs. We will present a linear time algorithm for

finding the next MPE. The l MPEs can be obtained by first finding the MPE and then calling the linear algorithm $l-1$ times to obtain next $l-1$ MPEs. For the sake of explanation, we define $D_a(b,c) = \Phi_a(a,b,c)$.

5.7.1 Sources of the next MPE

Having found the first MPE, we know the instantiated value of each variable and the associated instantiations of the other variables in the distribution in which the variable was reduced. If we replace that value with the second largest instantiation of the variable at the same associated instantiations of the other variables in the distribution, the result should be one of candidates for the second MPE. For example, assume $D_{a=A_1}(b=B_1,...,g=G_1)$ is the instantiated value for finding the first MPE. If we replace $D_{a=A_1}(b=B_1,...,g=G_1)$ with $D_{a=A_2}(b=B_1,...,g=G_1)$, the second largest instantiation of a, given the same instantiation of B through G, and re-evaluate all nodes on the path from that reduction operation to the root of the factor tree, the result is one of the candidates for the second MPE.

The total set of candidates for the second MPE comes from two sources. One is the second largest value of the last conformal product in finding the first MPE; and the other is the largest value of instantiations computed in the same computation procedure as for finding the first MPE, but replacing the largest instantiation of each variable independently where it is reduced with the second largest instantiation. The similar idea can be applied for finding the third MPE, and so on.

The candidates for finding the next MPE can be computed in the similar procedure as that in finding the first MPE, or obtained by searching the intermediate results for finding the first MPE with little computation. Considering the computational cost for the next MPE and the space cost for saving these intermediate results, we choose the searching strategy.

The factoring (or an evaluation tree) generated in step 2 of the algorithm in the previous section provides a structure for computing those candidates. We use the example in that section to illustrate the process.

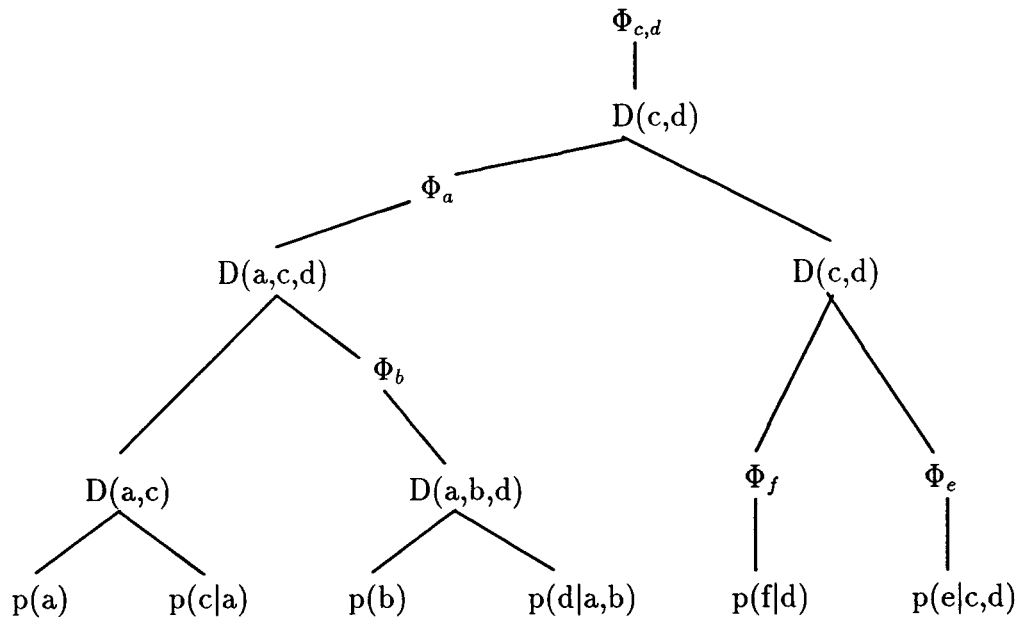


Figure 7. The evaluation tree for finding the MPE.

Figure 7 is the evaluation tree for finding the MPE for the belief network in the figure 5 section 5.2. Leaf-nodes of the evaluation tree are the original probability distributions of the belief network. The meaning of an interior node is the same as that which we used in previous sections. Two characteristics of the figure should be noticed. One is that an instantiated variable x only appears in the sub-tree rooted at the node Φ_x; so, changing instantiation of x happens only in this sub-tree. The other is that for any variables a and b, $D_a(b = B) \geq D(a = A, b = B)$, where A is any value of variable a except Φ_a, take $D_{a,b}(c, d) = \Phi_a D_b(a, c, d)$ as an example, then $D_{a=1,b=0}(c = 1, d = 1) \geq D_{\neg(a=1,b=0)}(c = 1, d = 1)$.

The two characteristics of an evaluation tree tell where and how to find the next largest value of an instantiated variable to replace its largest value for finding the next MPE. The following example gives the idea of the searching algorithm. In figure 7, since $d(c = 0, d = 0)$ of the node $D(c, d)$ (connecting to root node with $a = 0$, $b = 0$, $e = 0$ and $f = 1$) is the first MPE, if we find the second largest $d(c = 0, d = 0)$ (with another instantiation for variables a, b, e and f) to replace the

largest $d(c = 0, d = 0)$ in $D(c, d)$, the second MPE is the largest item in the $D(c, d)$. The second largest $d(c = 0, d = 0)$ results from either keeping $d(c = 0, d = 0)$ contributed from its left child node with no change, finding the second largest $d(c = 0, d = 0)$ from its right child node and multiplying the two values together or by keeping the $d(c = 0, d = 0)$ from its right child node, finding the second largest $d(c = 0, d = 0)$ from its left child node and multiplying them together. Then the problem of finding the second largest $d(c = 0, d = 0)$ is decomposed into the problem of finding the second largest $d(c = 0, d = 0)$ in each child node of the $D(c, d)$ node. As to the each child node, the same strategy can be used to find the desired candidate to contribute to the $D(c, d)$ node. This process may result in further decomposition of the original problem.

5.7.2 The algorithm for finding the next MPE

In order to efficiently search for the next MPE, we rearrange the computation results from finding the first MPE. The re-arrangement produces a new evaluation tree from the original evaluation tree, so that a sub-tree rooted at a node meets all constraints (variable instantiations) from the root of the tree to that node.

Evaluation Tree Re-arrangement The rules for converting the original evaluation tree to the new evaluation tree are as follows. If a node is $\Phi_{x,y,\ldots,z}$, duplicate the sub-tree rooted at the Φ node; the number of the sub-trees is equal to all possible instantiations of $\{x, y, \ldots, z\}$, and each sub-tree is constrained by one instantiation across $\{x, y, \ldots, z\}$. If a node is a conformal product node, nothing needs to be done. If a node has no Φ nodes in its sub-tree, prune the node and its sub-tree because all probabilistic information about the node and its sub-tree are known at its parent node. Figure 8 is an evaluation tree generated from the evaluation tree in figure 7. The evaluation tree in figure 8 is not complete; we only draw one branch of each Φ node.

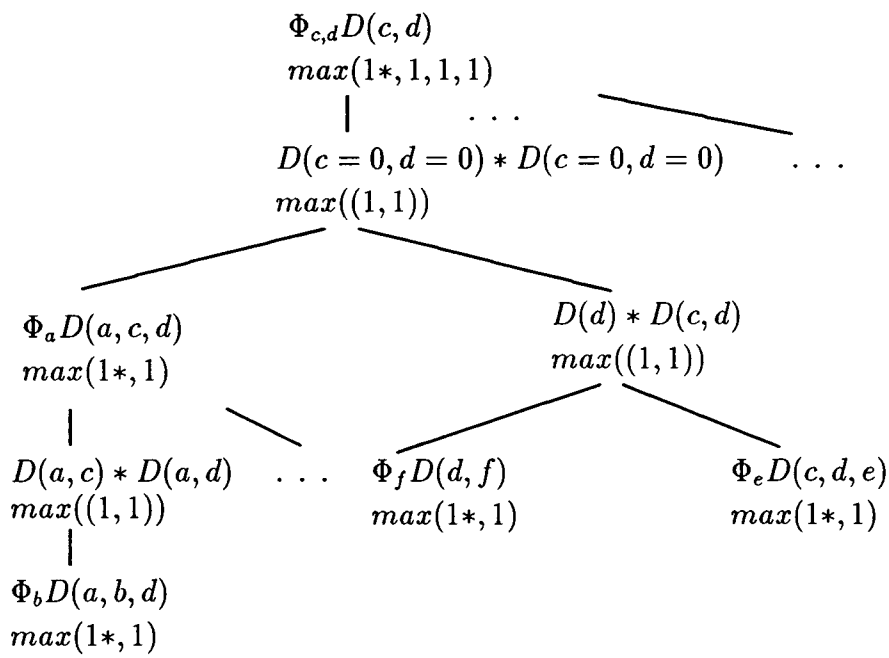


Figure 8. The evaluation tree for finding the next MPE.

Marking the Evaluation Tree The evaluation tree is annotated with marks to indicate the MPE's that have been returned. In figure 8 these marks are contained as the arguments to the *max* annotation at each node. There are two meanings for the parameters of *max*, depending on whether it is attached to a Φ or conformal-product node. An integer at a node denotes the ranking of the corresponding instantiated value contributed from its child node. For example, the first 1 at the root node indicates that the node contains the largest value of $d(c = 0, d = 0)$, and the "*" indicates that the value was used in a previous MPE (the first, in this case). The second 1 carries corresponding information for $d(c = 1, d = 0)$. For the conformal product immediately below the root node, the first 1 indicates the largest value of $d(c = 0, d = 0)$ has been retrieved from its left child node and the right 1 indicates the largest value of $d(c = 0, d = 0)$ has been retrieved from its right child node.

The Max Method The *max* method on an evaluation tree is defined as follows:

1. If a parameter is marked, i.e. its corresponding instantiated value was used for finding the previous MPE, generate the next instantiation - query (*max*) its child nodes to find and return the instantiated values matching the ranking parameters (we will discuss the determination of the parameters later).

2. If no parameter is marked, mark one parameter which corresponds to the largest instantiated value of the node, and return the value to its parent node.

The Gen Method We define a method *gen* to generate next ranking parameter for an integer i: $gen(i) = i + 1$ if (i+1) is in the domain, otherwise $gen(i) = 0$. The *gen* method for generating next possible ranking pairs of integers can be defined as follows. If current ranking pair is (i, j), then the next possible ranking pairs are generated:

1. If $(i - 1, j + 1)$ exists then $gen(i, j) = (i, j + 1)$;

2. If $(i + 1, j - 1)$ exists then $gen(i, j) = (i + 1, j)$.

The pairs $(0, x)$ and $(x, 0)$ exist by definition when x is in a valid domain size; *gen* will generate $(1, x + 1)$ and $(x + 1, 1)$ when applied to $(1, x)$ and $(x, 1)$. The range of an integer in a node is from 1 to the product of the domain size of these variables of Φ nodes in the sub-tree of that node. A pair of integer is valid if each integers in it is in the range.

The Algorithm for Finding Next MPEs Given the evaluation tree and the defined methods *max* and *gen* for each node, the procedure for finding the next l MPEs is: activate the *max* method of the root node l times.

We will retrieve the evaluation tree in figure 8 to see how the second MPE is found. This figure describes the initial status for finding the second MPE. The query for the second MPE is sent to the root node. Assume that the first 1 in the parameter list representing the value of $d(c = 0, d = 0)$ in the node is marked since the first MPE is $(a = 0, b = 0, c = 0, d = 0, e = 0, f = 1)$. In the step 1 of the *max* method, *gen* is activated to generate a new parameter, 2; and the new parameter is replaced for the marked parameter in the parameter list. Then, a query is sent to its child node for finding the second largest instantiated value for $d(c = 0, d = 0)$. If the child node returns the value, the *max* method will choose the largest value from all instantiated values of this node, mark the integer corresponding to the largest value, and return the value as the second MPE.

When the child node of the root node receives the query, it activates the *max* method to find the desired value. Since the only parameter $(1, 1)$ is marked, the *gen* method is activated for next possible ranking parameters; they are $(1, 2)$ and $(2, 1)$. This means that the candidates of the second largest instantiated value for $d(c = 0, d = 0)$ in the node come from either the product of the largest instantiated value from its left child node and the second largest instantiated value from its right child node or the product of the second largest instantiated value from its left child node and the largest instantiated value from its right child node. Since the largest instantiated values from both child nodes are known in the node, only the second largest instantiated value should be queried from each child node. Therefore, the *max* algorithm replaces $(1, 1)$ with $(1, 2)$ and $(2, 1)$, and queries its child nodes for the unknown ranking values. When both child nodes return the values, multiplications are carried out between those values according to $(1, 2)$ and $(2, 1)$. In step 2 of the *max* method, the larger value is chosen to return to its parent node and its corresponding parameter is marked.

Further, the queries from the second layer of the evaluation tree will go down to the third layer and so on, until a leaf node is met. A leaf node in the tree is a Φ type node in which the range for each parameter integer is just 1. Therefore,

gen will return 0 to replace the marked integer and set its corresponding value to 0. Then, the *max* method will choose the largest value and return to its parent node, and mark the corresponding integer parameter.

5.7.3 Analysis of the algorithm

The algorithm described above returns the next MPE every time it is called from the second MPE. First, we will show that the algorithm is complete; that is, it can find every possible instantiation of variables in a belief network. According to the rules for creating an evaluation tree, the number of different paths from the root to all leaves in the evaluation tree is equal to the product of the domain size of all variables in the belief network. That is, each path corresponds to an instantiation. Since the *max* method will mark each path it has retrieved during the finding of each successive MPE, and will not retrieve a marked path, the algorithm retrieves each path exactly once.

Second, the algorithm will always find the next MPE. When querying for the next MPE, the root node of the evaluation tree is queried to find a candidate which has the same instantiation for the variables in the root node as that for the previously found MPE, but has the next largest value. This computation is decomposed into the same sub-problems and passed to its child nodes, and from its child nodes to their child nodes, and so on. Each node being queried will return the next largest value to its parent node or will return 0 if no value can be found. Returning the next largest value from a node to its parent node is ensured by the *gen* and *max* methods. The *gen* method determines which instantiated value should be obtained from its child nodes. If the *gen* method has one integer as the parameter, it generates the successor of the integer or a zero as we expected. If the *gen* has a pair of integers as its parameter, we know, from the definition of the *gen* method, that the pair $(i, j+1)$ is generated only if $(i-1, j+1)$ exists; the pair $(i+1, j)$ is generated only if $(i+1, j-1)$ exists. On the other hand, if (i, i) is marked, it will not generate $(i, i+1)$ or $(i+1, i)$ unless $(i-1, i)$ or $(i, i-1)$ exist. Therefore, *gen* only

generates the pair needed for finding the next largest value in a node. Choosing the largest value from a list of instantiated values in *max* is obvious. From this we can conclude that the algorithm will always retrieve the next MPE each time it is called.

The time complexity of the algorithm for finding the next MPE in a belief network is linear in the number of instantiated variables in the evaluation tree. At a Φ node, only one marked value must be replaced by a new value. Therefore, only one child node of a Φ node needs exploring. At a conformal product node, there is at most one value to be requested from each child node according to the definition of *gen*. So, each child node of a conformal product node will be explored at most once. For example, after $gen(1,2)$ generates $(1,3)$, and $gen(2,1)$ generates $(2,2)$ and $(3,1)$, when $(2,2)$ is chosen, there is no query for $(2,2)$ because the instantiated values for $(2,2)$ can be obtained from $(1,2)$ and $(2,1)$ of previous computation. Therefore there are at most n Φ nodes plus $(n-1)$ conformal product nodes in an evaluation tree to be visited for finding the next MPE, where n is the number of nodes in the belief network. Also, there is a *max* operation in each node of the evaluation tree and only one or two multiplications needed in a conformal product node. Therefore, the algorithm for finding the next MPE is efficient.

The time complexity for converting a factoring to the evaluation tree for finding the next MPE should be no more than that for computing the first MPE. This conversion is the process of data rearrangement which can be carried out simultaneously with the process for finding the first MPE.

The space complexity of the algorithm is equal to the time complexity for finding the first MPE, since this algorithm saves all the intermediate computation results for finding the next MPE. The time complexity for finding the MPE in a singly-connected belief network is $O(k*2^n)$, where k is the number of non-marginal nodes of the belief network and n is the largest size of a node plus its parents in the belief network. Considering that the input size of the problem is in the order of $O(2^n)$, the space complexity is at most k times the input size for singly-connected

belief networks. For a multiply-connected belief network, the time complexity for finding the MPE can be measured by the maximum dimensionality of conformal products, which is determined by both the structure of a belief network and the factoring algorithm. The time complexity for finding the MPE in terms of input is exponential with respect to the difference between the maximum dimensionality for finding the MPE and the largest size of a node plus its parent nodes in the belief network. This time complexity reflects the hardness of the belief network if the factoring for it is optimal. If the factoring is optimal, the time and space complexity are the best that can be achieved for finding the l MPEs.

5.8 The MPE for a set of variables in belief networks

In this section, we will discuss the problem of finding the MPE for a subset of variables in belief networks. We will show that finding the MPE for a subset of variables in a belief network is similar to the problem of finding the MPE over all variables in the belief network, and the problem can be considered as an optimal factoring problem. Therefore, the algorithm for finding the MPE for a subset of variables in a belief network, either singly-connected or multiply-connected, can be obtained from the algorithm in section 5.2 with little modifications.

We first examine the differences between probabilistic inference (posterior probability computation) and finding the MPE for all variables in a belief network so that we can apply the approach described in section 5.2 to the problem of finding the l MPEs for a subset of variables. There are three differences. First, there is a target or a set of queried variables in posterior probability computation; but there is no target variable in finding the MPE. The computation for a posterior probability computation is query related and only the nodes relevant to the query are involved in the computation, whereas finding the MPE relates to the whole belief network. Second, the addition operation in summing over variables in posterior probability computation is replaced by comparison operation in finding the MPE,

but the number of operations in both cases is the same. And finally, variables with no direct descendants in a distribution can be reduced at the beginning of finding the MPE whereas queried variables cannot be summed over in posterior probability computation.

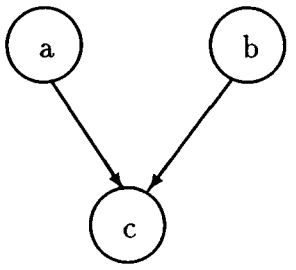
Finding the MPE for a set of variables in belief networks combines elements of the procedures for find the MPE and for posterior probability computation. Since not all variables in a belief network are involved in the problem of finding the MPE for a set of variables, the variables not relevant to the problem can be eliminated from computation. Therefore, two things should be considered in finding the MPE for a set of variables in a belief network. One thing is to choose relevant nodes or distributions for computation. The second is to determine the situation in which a variable can be summed over or reduced. The first is simple because we can find the relevant nodes to some queried nodes given some observed nodes in linear time with respect to the number of nodes in a belief network[GVP89, SDD90]. We have the following lemmas for determining when a node can be summed over or reduced.

Suppose we have the variables relevant to a set of queried variables for finding the MPE given some observations. These variables can be divided into two sets: a set Φ which contains the queried variables (or the target variables for finding the MPE) and a set Σ which contains the rest of the variables (or variables to be summed over in computation). The current distributions are represented by D_i for $1 \leq i \leq n$ and the variables in a distribution D_j are also represented in the set D_j.

Lemma 5.3 *Given $\alpha \in \Sigma$, if $\alpha \in D_i$ and $\alpha \notin D_j$ for $i \neq j$, $1 \leq j \leq n$, then α can be summed over from the distribution D_i.*

Proof: The lemma is obvious. It is the same situation in which we sum over some variables in posterior probability computation. ∎

Lemma 5.4 *Given $\alpha \in \Phi$, if $\alpha \in D_i$ and $\alpha \notin D_j$ for $i \neq j$, $1 \leq j \leq n$, and for any other $\beta \in D_i$, $\beta \in \Phi$, then the α can be instantiated and eliminated for the distribution D_i.*



p(a=1)=0.9
p(b=1)=0.1
p(c=1|a=1,b=1)=0.6
p(c=1|a=1,b=0)=0.8
p(c=1|a=0,b=1)=0.3
p(c=1|a=0,b=0)=0.4

Figure 9. A simple belief network for finding the MPE for nodes b and c.

Proof: Since $\alpha \in \Phi$ and $\alpha \in D_i$ only, the information relevant to α is in the distribution D_i. So, we can instantiate variable α to find its largest instantiated value to contribute the MPE, and the reduced distribution of D_i contains all possible combinations across values of other variables in D_i. Since for any other $\beta \in D_i$, $\beta \in \Phi$, no summation for some other variables of D_i afterward will affect the β. So β can be instantiated later if possible. ∎

We can show, by an example, that if the condition, for any other $\beta \in D_i$ and $\beta \in \Phi$, is not satisfied, we can not apply the lemma 5.4 for instantiating a variable in a distribution. We want to find the MPE for nodes b and c in the belief network in figure 9. In the step 1 of the algorithm in section 5.2, the distribution $p(c|a,b)$ can be reduced to $D(a,b)$: $D(a=1,b=1)=0.6$ with $c=1$, $D(a=1,b=0)=0.8$ with $c=1$, $D(a=0,b=1)=0.7$ with $c=0$ and $D(a=0,b=0)=0.6$ with $c=0$. The conformal product of $D(a,b) * D(a)$ generates $D(a,b)$: $D(a=1,b=1)=0.54$ with $c=1$, $D(a=1,b=0)=0.72$ with $c=1$, $D(a=0,b=1)=0.07$ with $c=0$ and $D(a=0,b=0)=0.06$ with $c=0$. We need to sum over variable a in distribution $D(a,b)$; but we get a problem. We can not add $D(a=1,b=0)=0.72$ with $c=1$ with $D(a=0,b=0)=0.06$ with $c=0$ together because of different c values. The problem was caused by instantiating variable c in distribution $p(c|a,b)$ when variable $a \notin \Phi$.

Given the two lemmas, the algorithm in section 5.2 can be modified for finding the MPE for a set of variables in belief networks. Given a belief network, a

set of variables Φ and evidence variables E, the algorithm for finding the MPE of Φ is:

1. Find variables of T which are the predecessors of variables in set Φ or E and connected to set Φ[16]. The distributions relevant to the variables in T are needed for finding the MPE of Φ.

2. For any variable x of T having no descendants in the belief network, reduce the conditional distribution of the node x by choosing the items of the distribution which have the largest instantiated values of x with same associated instantiations for the other variables. The reduced distribution has no variable x in it.

3. Create a factoring for all distributions.

4. Combine these distributions according to the factoring. Apply lemma 5.3 and lemma 5.4 to each result distribution in probability computation. If both lemmas apply to a distribution, apply lemma 5.3 first.

Take the belief network in figure 4 as an example. We want to find the MPE for the variables $\Phi = \{c, d, e\}$ given E is empty. In step 1 of the algorithm, the variables related to the query are found, $T = \{a, b, c, d, e\}$. In step 2, distribution $D(c, d, e)$ is reduced to $D(c, d)$. In step 3, assume a proper factoring is found:

$$((D(a) * D(a, c)) * (D(b) * D(a, b, d))) * D(c, d).$$

In step 4, combine these distributions according to the above factoring and apply lemma 5.3 or/and lemma 5.4 to any result distribution if applicable.Then we obtain the MPE for variables $\{c, d, e\}$. The whole computation can be represented as:

$$\Phi_{c,d}(\sum_{a}((p(a) * p(c|a)) * (\sum_{b}(p(b) * p(d|a, b)))) * \Phi_e p(e|c, d)).$$

[16]An evidence node breaks the connection of the node with its child nodes.

This algorithm is very similar to the algorithm in section 5.2. Since the time complexity of the first step of the algorithm is linear with respect to the number of variables in belief networks, the most time consuming step of the algorithm is step 4 which is determined by the factoring result of step 2. Therefor, efficiently finding the MPE for a set of variables in a belief network can be considered as an optimal factoring problem. By using the algorithm presented in the previous section after finding the first MPE, the problem of finding the l MPEs for a set of variables can be easily solved.

In this section we have presented an algorithm for the problem of finding the MPE for a set of variables in a belief network and shown that the problem can be efficiently solved through an optimal factoring problem. But, we didn't present a factoring algorithm here. We have discussed the difference between this problem and the problem of finding the MPE for all variables in a belief network, and the difference between this problem and the problem of computing posterior probability of a set of variables. So, we can apply the factoring strategies developed in the previous chapter for posterior probability computation or for finding the MPE for a whole belief network to this problem. Of course, we can design a more efficient factoring algorithm for the problem itself. However, we will not discuss this further or present any algorithm for the problem in this paper.

5.9 Related work

Dawid [Daw92] pointed out that the problem of finding the MPE of a belief network can be simply realized by replacing the normal marginalization operation of the distribution phase of evidence propagation in a join-tree in posterior probability computation by max-marginalization (i.e. taking max instead of summing). Therefore, the efficiency of an algorithm for finding the MPE depends basically on the corresponding posterior probability computation algorithm. Golmard developed an

algorithm for finding the MPE independent of our work [Gol92]. We have requested a copy of the work and are waiting to receive it.

The problem of finding the MPE is similar to the problem of querying marginal, conditional or conjunctive queries in a belief network. There are many other exact methods, besides the factoring method, for probabilistic inference in belief networks. Those methods can be found in [Sha86, Pea88, Sha88, LS88, Hec89, SKA89, D'A89, SDD90, JOA90, Coo90b, SAS91a], and so on. Shachter has shown that those methods or derived algorithms form those methods, except the factoring method, are equivalent to a clustering algorithm [SAS91a]. The factoring method has some advantages over the other methods. For example, one of its advantages over a clustering algorithm, in [Pea88] and so on, is that the factoring method has not necessarily been constrained by the triangulation rule for combining nodes in each case. This topic will be discussed in more detail later.

The factoring method proposed in this paper converts the problem relevant to networks or graphs into a numeric optimization problem. Related work in numeric optimization can be found in [NW88, PS82, Hu82, LC78, Pea84, Sni92]. The work relevant to the optimization with networks can be found in [ST63, Ros73, Wen90] etc.

5.10 Conclusion

In this chapter we have presented a framework, optimal factoring, for finding the most probable explanations (MPE) in a belief network. Under this framework, efficiently finding the MPE can be considered as the problem of finding an ordering of distributions in the belief network and efficiently combining them. The optimal factoring framework provides us with many advantages for solving the MPE problem. First, the framework reveals the relationship between the problem of finding the MPE and the problem of querying posterior probability. Second, quantitative description of the framework provides a way of measuring and designing an

algorithm for solving the problem. Third, the framework can be applied to both singly-connected belief networks and multiply-connected belief networks. Fourth, the framework can be applied to the problem of finding the MPE for a set of variables in belief networks. Finally, the framework provides a linear time algorithm for finding the next MPE. Under the optimal factoring framework, we have developed an optimal factoring algorithm for finding the MPE for a singly-connected belief network. We have also developed an efficient algorithm for finding the MPE in multiply-connected belief networks.

Chapter 6

Parallelizing Probabilistic Inference

6.1 Motivation

The cost of a conformal product in probability computation is exponential with respect to the number of variables in the conformal product. This renders any currently used exact probabilistic inference algorithm intractable for large belief networks, with more than 100 nodes and more than five arcs per node. One way to extend this boundary is to consider parallel hardware.

The feasibility of parallelism in probabilistic inference was first experimentally tested through simulation with a hypercube architecture for the Symbolic Probabilistic Inference algorithm (SPI)[Fou91][17]. The results presented there indicated that in a shared memory model, reasonable speedup can be achieved; in a distributed memory model, however, little speedup is available.

The performance of parallel probabilistic inference in a distributed memory model is determined by the following factors:

1. parallel model or parallelism for probability computation;

2. parallel architecture;

[17]We haven't seen the exploration of any other algorithms for parallelizing probabilistic inference.

3. parallelizability or inherent communication complexity of probabilistic inference; and

4. computing strategy for each query.

We do not think that the poor performance was caused by the first factor above, because the good speedup with the shared memory model reflected the properly chosen parallel model in probability computation. Since the performance is different between the shared memory model and distributed memory model with the same parallel architecture, we think the poor performance was caused by either the computing strategy or the inherent complexity of communication in parallel probabilistic inference.

Changing the computing strategy is a simple way of exploring the feasibility of parallelizing probabilistic inference. A computing strategy determines the way of combining distributions in parallel probabilistic inference. Therefore, to find an appropriate computing strategy is to find a good factoring for a query. The goal of our research is to find new factoring algorithms for parallel probability computation.

In this chapter, we will analytically explore the problem of parallel probabilistic inference within a hypercube architecture. We consider efficient parallel probabilistic inference in belief networks as a combinatorial optimization problem, in particular as an optimal factoring problem, and present a new approach to solve the problem. Optimal factoring considers the global information from a combinatorial optimization point of view and tries to find an optimal combination for all distributions. We can, from this point of view, explore the characteristics of efficient parallel probability computation, explain the reason of poor performance of parallel probability computation for SPI, and easily find efficient algorithms for parallel probability computations. The optimal factoring perspective provides a way of considering communication cost in an algorithm design.

The remainder of the chapter is organized as follows. Section 2 discusses the parallelism for probability computation. Section 3 presents a simulation model for

the parallel architecture we used in parallelizing probabilistic inference. Section 4 introduces a combinatorial optimization problem, optimal factoring, for parallelizing probabilistic inference. Section 5 discusses the characteristics of parallelizing probabilistic inference in belief networks from OFP perspective. Section 6 presents a heuristic factoring algorithm we have developed for parallel probability computation. Section 7 shows an experimental test of the algorithm. Finally, section 8 summarizes the research.

6.2 Parallelism in probability computation

From chapter 3 we know that different algorithms for probabilistic inference are just different factoring strategies. The function of these algorithms is to create an evaluation tree for a query and evaluate it. Take the belief network in Figure 4 as an example. We want to query the probability of the node e with observation $f = 1$, i.e.. $p(e|f = 1)$. One of the possible ways of computing the query is:

$$p(e) = \sum_{a,b,c,d,} ((p(e|c,d) * (p(c|a) * p(a))) * (p(f = 1|d) * (p(d|a,b) * p(b)))).$$

The computation of the above formula corresponds to an evaluation tree in which the leaves are the probability distributions of some variables relevant to the current query, and each non-leaf node represents a conformal product of two distributions represented by its two children and gives the result distribution of the multiplication.[18] For the query $p(e|f = 1)$, its evaluation tree is given in figure 10.

Parallelism in probability computation can be carried out at two levels: one is *evaluation-tree parallelism* and the other is *conformal product parallelism.* Evaluation-tree parallelism is a kind of control parallelism which computes different conformal products in parallel. Take Figure 10 as an example: the evaluation-tree parallelism is the computation of two branches from the root node simultaneously.

[18]It may include some addition operations if some variables would be summed over after multiplication.

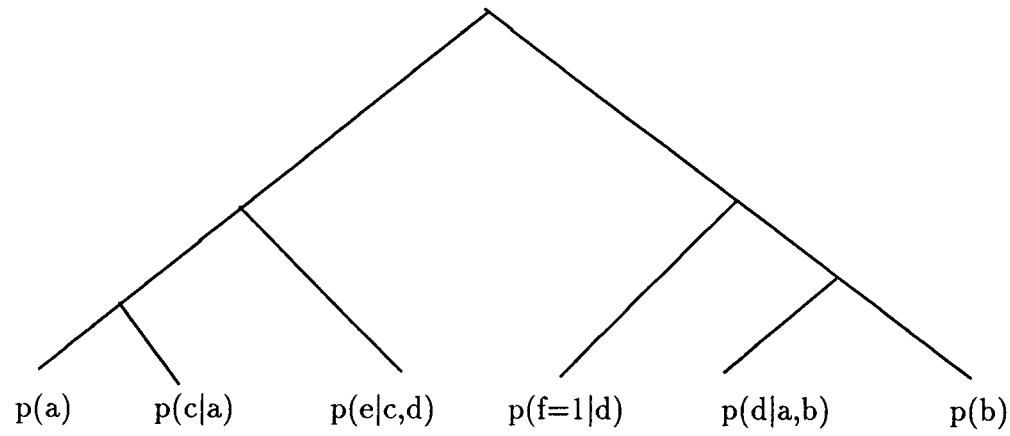


Figure 10. The evaluation tree of querying p(e) given $f = 1$ in the simple belief network.

Conformal product parallelism is a kind of data parallelism which dispatches a conformal product among a group of processors and decreases the number of variables appearing in the computation at each processor. The parallelism can be carried out by distributing data to each processor according to the values of some of variables, called *splitting variables*, in the two distributions. A simple example of conformal product parallelism is to consider the conformal product $p(d|a,b) * p(b)$ in figure 10. Its parallel computation by two processors is shown in figure 11. The data are split into two portions and sent to each processor according to $b = 0$ or $b = 1$, and the result from each processors is returned to one processor where all results are assembled to form a final result. The cost of assembling the final result is relatively small compared with the cost of the conformal product. We will ignore the cost in assembling data in this paper. However, the communication cost for returning the data cannot be neglected.

Since each non-leaf node in an evaluation tree corresponds to a conformal product operation and distributions occur at most once in the evaluation tree (i.e., it is a tree, and therefore singly-connected), it is possible to parallelize a conformal product operation corresponding to the nodes in disjoint subtrees, and therefore the evaluation tree is parallelizable too.

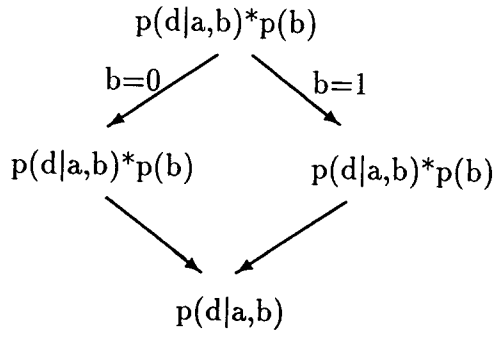


Figure 11. The conformal product parallel computation with 2 processors.

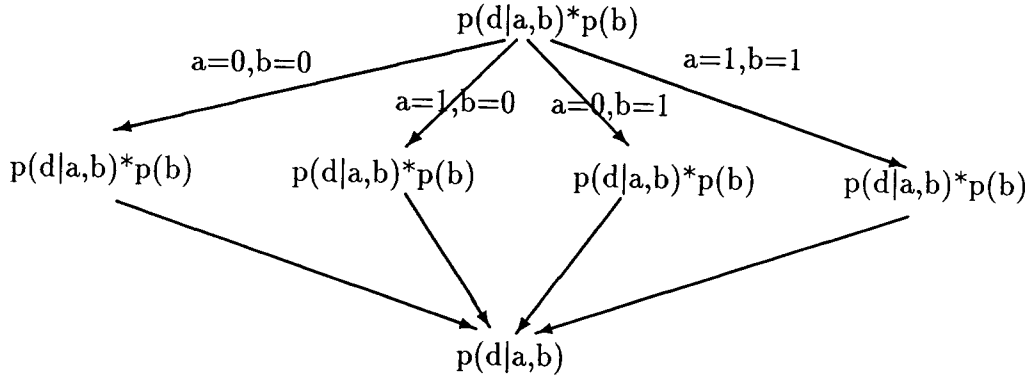


Figure 12. The conformal product parallel computation with 4 processors.

In conformal product parallelism, two factors should be considered [Fou91]. One is which variables should be chosen as splitting variables and the other is how many variables should be chosen as splitting variables. Splitting more variables in a probability computation results in more parallelism but increases the total amount of data to be distributed for the computation. Figure 11 shows the result of one variable b being chosen as a splitting variable and in figure 12 two splitting variables were chosen instead. In the paper [Fou91], the effect of these factors on the total amount of data to be sent to each processor was studied. It was found that splitting input distributions on variables that occur in both input distributions results in no increase in the total amount of data communication required to compute the conformal product, and that splitting the input distributions on variables other than those in the result distribution requires either the ability to perform concurrent writes with summing or a separate processing step in which the values that make up the result distribution entries are summed together. In [Fou91], the splitting variables were chosen only from result distribution, and the variables appearing in both distributions were chosen in the test. We will see later that this strategy should be changed in some cases.

The evaluation-tree parallelism can be estimated by computing the ratio of the amount of work used for the longest path or costliest path in an evaluation tree with the total amount of work in the whole tree. The effect of evaluation tree parallelism depends on the structure of an evaluation tree.

Two other factors that influence the speed of a parallel computation are minimum task grainsize and the number of processors available. Reasonably choosing the two parameters is important in parallel probability computation. The number of processors available is chosen as 1024 here. The determination for choosing the minimum task grainsize will be discussed later.

6.3 Models

The parallel model of computation used for this investigation was based on a broadcast-compute-aggregate (BCA) model of conformal product evaluation [DFL92].. Under this model a distinguished processor is responsible for partitioning each conformal product into subtasks, which are distributed to the other processors for evaluation. This same distinguished processor collects the results from the subtasks and assembles them into the result distribution. This model is communication-intensive, since all data originates and returns to a single processor. An alternative to the BCA approach is the distributed-net model (Dist-net), in which the entire belief net, including all intermediate results, is stored on each processor. Compared to the BCA model, the Dist-net model trades memory for communication costs. For this chapter we concentrate our discussion on the BCA model. We do, however, present some preliminary results at the end of this chapter comparing the Dist-net and BCA models.

We use two-level models of sequential and parallel inference in this chapter. First, we use a low-level model of individual conformal-product evaluation. Second, we use a higher-level model of query-tree evaluation. Each of the models was parameterized to allow experimentation with various aspects of the computational environment including task grainsizes, number of processors, and communication costs.

6.3.1 Sequential Model

To provide a basis against which to measure the performance of the parallel algorithms, we developed the following model to estimate the running time of the sequential algorithm.

Sequential conformal product model: $T_s(c) = \alpha M(c)$

where $M(c)$ is the number of multiplies in conformal product c and α is a constant scaling factor that accounts for factors other than multiplies in the conformal

product operation, most notably indexing and additions. $M(c)$ is calculated by $\prod_{v \in V} n(v)$, where V is the union of the variables occurring in the two input distributions. To simplify analysis, we restricted the value space of the variables in the test belief nets to 2 values each. Thus in the following sections $M(c) = 2^n$ where n is the number of unique variables in the 2 input distributions.

Given the model for the sequential running time for a single conformal product, the estimated sequential running time for the evaluation of an entire query is the sum of the time required for each of its conformal products.

Sequential model for a query: $T_s(q) = \sum_{c \in q} T_s(c)$

6.3.2 Conformal Product Parallel Model

The following model was developed to estimate the running time of a parallel algorithm.

Parallel conformal product model: $T_p(c) = P + S + W + C$. Where
P is the cost for process initialization,
S is the cost for setting the problem up,
W is the cost for the work done at each processor, and
C is the cost for communication.

The value which was used for W was αG, where G is the computational grainsize, specified as the number of floating point multiplies per process; and α is the same constant scaling factor that was used in the sequential model.

The following measures were calculated for each conformal product:

T_s = Time for sequential

T_p = Time for parallel

N_u = Number of processors used;

The values for G and N_u were determined as follows:

1. A minimum grainsize, G_{min}, and a maximum number of processors, N_a, were specified.

2. Given a particular conformal product to compute, the actual G and N_u values were calculated so that N_u was maximized under the constraints $N_u \leq N_a$, $G \geq G_{min}$, and $N_u \leq 2^v$, where v is the number of variables in the result distribution. In other words, N_u and G were chosen such that as many of the available processors as possible were used, as long as there was enough work for each processor to perform as specified by the minimum grainsize, and there was enough parallelism in the problem to support the desired partitioning.

Distributed-Memory Communication Model

The distributed-memory model includes a specific model of communication for a cube architecture. This model assumes that there is no overlap between the conformal product calculations and the communication between processors. This model also assumes that the data to be sent to processors is arranged into buffers, one buffer for each process.

Total communication cost: $C = C_d + C_r + B$

where

C_d is the communication cost for distributing the data,

C_r is the cost of returning the data, and

B is the cost for building the buffers of data to be distributed.

The data transfer cost calculations were based on the log spanning tree, or broadcast, communication model for hypercube [Fox88].

Distribution communication cost: $C_d = (D_{max} * C_{st}) + (B_d * ((N_u - 1) * C_b))$

where

D_{max} is the maximum dimension of the cube which is used,

C_{st} is the communication startup time,

B_d is the number of bytes sent to each processor, and

C_b is the communication cost per byte, per link.

The following formula was used to calculate B_d, the number of bytes sent to each processor:

$B_d = B_{total}$ / N_u;

where B_{total} is the total number of bytes needed to compute the conformal product. B_{total} is, in general, greater than the sum of the sizes of the two input distributions.

Assuming 4 bytes per word, a lower bound for B_{total} can be calculated by the following formula:

$$B_{total} = 4 * (2^{max(d1,s)} + 2^{max(d2,s)}),$$

$$B_{result} = 4 * 2^r;$$

where |d1| and |d2| represent the number of variables in the 2 input distributions and |s| represents the number of result distribution variables on which splitting occurs.

We consider the worst case of B_{total} in the model as in [Fou91]. This makes it easier for us to compare the analytical results with some experimental results made before [Fou91, DFL92]. This method is quite conservative. If this model demonstrates the feasibility of parallel probability computation, other more accurate models should predict better performance.

The return communication cost is calculated in the same way as distribution communication costs:

Return communication cost: $C_r = (D_{max} * C_{st}) + (B_r * ((N_u - 1) * C_b))$

where

B_r is the number of bytes returned from each processor and is calculated by

$B_r = B_{result}$ / N_u

where B_{result} is the total number of bytes in the result distribution. Since there are $2^{|result-dist-vars|}$ entries in the result distribution, $B_{result} = 4 * 2^{|result-dist-vars|}$.

6.3.3 Parallel Model of Query Evaluation

As explained earlier, query evaluation consists of repeated conformal product operations. Since we were interested in the performance of a parallel inference algorithm

on the task of query evaluation, we constructed a model to predict this performance from the models for conformal product operations. The parallel query model is analogous to the sequential query model. The running time for parallel query evaluation is simply the sum of the running times of its conformal products.

Parallel model for a query: $T_p(q) = \sum_{c \in q} T_p(c)$

$N_u(q)$, the number of processors used in evaluating query q, is simply the maximum of the number used by any of the conformal products in q. Given $T_p(q)$, $T_s(q)$ and $N_u(q)$ the speedup, cost, and efficiency for a query can be calculated according to formulas given in [Akl89].

Speedup: $S(q) = T_s(q)$ / $T_p(q)$

Cost: $C(q) = T_p(q)$ * $N_u(q)$

Efficiency: $E(q) = T_s(q)$ / $C(q) = S(q)$ / $N_u(q)$

6.3.4 Evaluation Tree Parallelism Models

For evaluation tree parallelism, we only computed a lower bound on running time. As mentioned earlier, a lower bound on the running time of an algorithm exploiting evaluation-tree parallelism can be calculated by summing the times required to perform each of the conformal products in the longest path of the evaluation tree.

Lower bound on $T_p(q) = \sum_{c \in Lp} T_s(c)$ where Lp is the longest path in the evaluation tree as measured by the amount of time it takes to compute the conformal products in the path.

6.3.5 Model initiative

Since we used a model rather than an actual parallel implementation, we had to make assumptions about the number of processors available, processor speeds, communication costs, and so on for the analyses and experimental tests in the following sections. We assumed a maximum of 1024 processors, and chose a minimum task grainsize of 256. We set α, the scaling factor for multiplication, at 45 microseconds.

P, the cost for process initialization, was taken to be 0. S, the cost for setting the problem up, was 0. C_{st}, the communication start-up time, was 230 microseconds. C_b, the communication cost per byte, was 0.5 micro seconds per byte per link. B, the cost for building the buffers of data to be distributed, was 0. These values are believed representative of actual costs on an Intel IPSC-2, and are based on discussions with the parallel algorithms and languages groups at OSU. Further, the relative values of these numbers seem to be valid for announced and foreseeable hypercube-style machines. Since our speedup measurements are dependent on the relative values, rather than the absolute values, we expect that our results are applicable to most machines in this architecture class. For further discussion see [Fou91].

6.4 Optimal factoring problem

Given the two parallelisms in parallel probability computation, we know that parallelizing probabilistic inference can also be considered as a factoring problem. In fact, in both sequential and parallel cases, we need to create an evaluation tree for a query. The difference lies in the way of counting cost of computation. In chapter 3, we have defined the optimal factoring problem for sequential probability computation. Similarly, we can define an optimal factoring problem for parallel probability computation. Without loss of generality we assume that the domain size of each variable is 2. The problem can be described as follows.

Definition [PFP] *Given*

1. *a set of m variables V,*
2. *a set of n subsets of V:* $S = \{S_{\{1\}}, S_{\{2\}}, ..., S_{\{n\}}\}$, *and*
3. $Q \subseteq V$ *is a set of target variables*

define operations:

1. *combination of two subsets S_I and S_J:*

$$S_{I\cup J} = S_I \cup S_J - \{v : v \notin S_K \text{ for } K \cap I = \phi,$$

$$K \cap J = \phi, \text{ and } v \notin Q\},$$

$$I, J \subseteq \{1, 2, ..., n\},\ I \cap J = \phi;$$

2. *the cost function of combining the two subsets:*

$$\mu(S_{\{i\}}) = 0 \text{ for } 1 \leq i \leq n, \text{ and}$$

$$\mu(S_{I\cup J}) = max(\mu(S_I), \mu(S_J)) + C_{cmp}(S_I, S_J, N) + C_{cmm}(S_I, S_J, N).$$

The factoring problem is to find a way to combine the n subsets.

In the above definition, Q is a set of target variables after combining all subsets of S together; the set $\{v\}$ in the formula $S_{I\cup J}$ are the variables which do not appear in the remaining subsets of S after combination of S_I with S_J and do not appear in the set Q either. $C_{cmp}(S_I, S_J, N)$ is the computation cost of the combination of sets S_I with S_J on one processor with N processors used. $C_{cmm}(S_I, S_J, N)$ is the communication cost of the combination of sets S_I with S_J for distributing data to N processors and the communication cost for the results' integration. The first operation describes a way of combining two sets together to form a new set. The second operation measures the cost of combining the sets whose index are in the set $(I \cup J)$. When $(I \cup J)$ includes 1, 2,..., n, $\mu_\alpha(S_{I\cup J})$ is the total cost of combining all set S_i ($i \in I, J$) in a given factoring order.

$\mu(S_I)$ is not unique if $|I| > 2$. In general, it depends on how we combine the subsets. We indicate these alternative combinations by subscripting μ. $\mu_\alpha(S_I) = \mu$ shows the cost of combining the result of S_I with respect to a specific tree structured combination of I, labeled α. We call this combination a factoring.

Definition [POFP] *A* **parallel optimal factoring** *problem is to find a factoring α such that $\mu_\alpha(S_{\{1,2,...,n\}})$ is minimal.*

The difference between the sequential factoring problem and the parallel factoring problem is that in the sequential OFP, only the computation cost exists in set combination; while in the parallel POFP, both computation cost and communication cost exist in set combination. Also, the computation cost in sequential computation is determined by the sets to be combined, while the computation cost and communication cost in parallel POFP are determined by the sets to be combined, by a parallel architecture and by the number of processors available in parallel computation. As a result, we can't give an explicit expression for the cost in the POFP definition without knowing the parallel architecture. Comparing the sequential OFP with the parallel POFP, we know that the parallel POFP is a problem at least as hard as sequential OFP, which, we believe, is an NP-hard problem[19]. Therefore, to solve the parallel POFP, we will not try to find a general optimal factoring algorithm, instead, we are trying to find a effective heuristic algorithm for the problem.

The definition of the POFP provides a way of quantitatively analyzing parallel combination cost. The purpose of defining the POFP is to find a way to analyze parallel probability computation. If we can map parallel probability computation problem to POFP, we can consider the problem algebraically instead of graphically. Graphical representation usually provides an intuitive and explicit way of understanding relations of variables in a problem while we find an algebraic representation more suitable for analysis and algorithm generation. In the next section we will discuss the mapping between the problem of parallelizing probabilistic inference and the POFP. As an POFP, we can look at the problem of parallel probability computation more adequately, and easily explore the essence of the problem.

[19]We have not yet proven the complexity of the sequential OFP. From the similarity between the problem and the traveling salesman problem, we believe the sequential OFP is NP-hard.

6.5 Parallel probability computation as an POFP

Given a belief network with observations, not all the nodes in the belief network are relevant to all queries. The nodes related to the query correspond to a new graph which is a sub-graph of the original belief network. The sub-graphs relevant to different queries, given different observations in a belief network, may vary tremendously. Therefore, the physical structure of the original belief network, the number of nodes and the average arc per node in the belief network does not always reflect the characteristics of each query. Since the minimum number of conformal products for a query in a belief network is unique and the maximum dimensionality of the conformal products reflects the computational complexity of the query for a particular algorithm, it is reasonable to consider the number of conformal products and the maximum dimensionality of the conformal products as the parameters to describe a query. Furthermore, since one or a few conformal products with maximum dimensionality dominate the total computation cost, the number of conformal products is not critical. So, we simplify our consideration to the conformal product with the maximum dimensionality in conformal product parallelism.

The problem of efficiently parallelizing probabilistic inference in a belief network can be mapped to the POFP. Given a belief network with m nodes and some observations in it, a query involves identification of a subset of n nodes relevant to the query in the belief network of m nodes and computation of the conformal product [SDD90] of marginal and conditional probabilities of the n nodes. The n nodes with their relations can be mapped to the symbols in the definition for POFP: the n nodes with their immediate antecedent nodes are mapped to the n subsets of m variables; the queried nodes correspond to the variables in the subset Q.

The Parallel Optimal Factoring for a query contains an evaluation tree which has the minimum cost in parallel computation for that query. After the mapping to an POFP, the set combinations represents conformal products; $S_{I \cup J}$ denotes the result variables after the conformal product; $C_{cmm}(S_I, S_J, N)$ and $C_{cmp}(S_I, S_J, N)$

are the cost of a parallel conformal product S_I with S_J; the term $max(\mu(S_I), \mu(S_J))$ is the result of evaluation-tree parallelism which chooses a branch of the tree rooted at the interior node $S_{I\cup J}$ with the most cost; and $\mu(S_{I\cup J})$ gives the minimum cost of parallel probability computation for the sub-tree rooted at the interior node $S_{I\cup J}$. When $(S_{I\cup J})$ covers all distributions, the result $\mu(S_{I\cup J})$ is the cost of the query in the belief network. It should be clear that finding a lowest cost way of computing the evaluation-tree is the problem of POFP.

In a sequential OFP, the number of multiplications for current combination is $2^{|S_I\cup S_J|}$. If we consider the α value approximating to 2^6 in the defined parallel model, the equivalent computation cost in parallel computation, in micro seconds according to the parallel model, is:

$$C_{seq} = \alpha 2^{|S_I\cup S_J|} \approx 2^{md+6} \tag{6.15}$$

where md represents $|S_I \cup S_J|$.

In parallel POFP, if the computation can be distributed to N processors, then

$$C_{cmp}(S_I, S_J, N) = \alpha 2^{|S_I\cup S_J|}/N = \alpha 2^{|S_I\cup S_J|}/2^n \approx 2^{md-n+6}, \tag{6.16}$$

if we assume $N = 2^n$. We call the n the *processor dimension* hereafter.

The communication cost for the combination depends on a parallel computation model. Given the parallel computation model in section 6.2, we can estimate the communication cost in the model. The communication cost of C_d and C_r are

$$C_d = (D_{max} * C_{st}) + 2 * (2^{max(d1,s)} + 2^{max(d2,s)}) * \frac{(N-1)}{N} \tag{6.17}$$

$$C_r = (D_{max} * C_{st}) + 2 * (2^r) * \frac{(N-1)}{N} \tag{6.18}$$

Considering an adequate N value, C_d and C_r are approximately equal to:

$$C_d \approx (D_{max} * C_{st}) + 2 * (2^{max(d1,s)} + 2^{max(d2,s)}) \tag{6.19}$$

$$C_r \approx (D_{max} * C_{st}) + 2 * (2^r) \tag{6.20}$$

Then the total communication cost is:

$$C_{cmm} \approx 2 * (D_{max} * C_{st}) + 2 * (2^{max(d1,s)} + 2^{max(d2,s)} + 2^{r}). \quad (6.21)$$

From the formulas above, we can numerically analyze the parallel computation model and derive some results.

Splitting variables. We consider the case that a conformal product can be fully parallelized here. That is, all variables in the result distribution can be used as splitting variables and the processor dimension is greater than the number of splitting variables, namely $n > s$. Then, the total communication cost is

$$C_{cmm} \approx 2 * (D_{max} * C_{st}) + 2 * (2^{max(d1,s)} + 2^{max(d2,s)} + 2^{s}), \quad (6.22)$$

and the computation cost is about

$$C_{cmp} \approx 2^{md-s+6}. \quad (6.23)$$

From the formulas 6.22 and 6.23 we can see that C_{cmp} dominates the value of total cost when s is small, called *under splitting*. Here, we assume 2^{md} is much bigger than $(D_{max} * C_{st})$; otherwise, we can solve the problem quickly in a single processor. Thus increasing s will decrease the value of total cost until C_{cmm} is greater than or equal to C_{cmp}. This indicates that the number of variables chosen as splitting variables should not depend on the number of variables in the result distribution when under splitting happens, i.e., when the number of variables in the result distribution is less than the difference of the dimensionality of the conformal product with the processor dimension. In this case, variables not in the result distribution should be chosen as splitting variables, but with the paying off in the cost by data dependence and assembling final results. Since the cost is small compared with the exponential computation cost, it shouldn't be a problem. The variables appearing in both distributions are chosen first in this case.

Grainsize. From formula 6.22, we also know that C_{cmm} is at least $2*(D_{max}*C_{st})$. If we want to get a positive parallel result, C_{cmp} should be at least greater than the $(D_{max} * C_{st})$. That is $md - s + 6 > 13$, if we consider the number of processors is 1024 and C_{st} is 230 in the parallel model. If $s = 0$, then $md > 7$. This indicates that the grainsize should be at least greater than 2^7 for this parallel model. In the experiment test [DFL92], the grainsize was chosen 2^8.

Speedup in conformal product parallelism. From formulas 6.15, 6.16, 6.17 and 6.18, the relative speedup of parallel computation for a conformal product is

$$rspd \approx \frac{2^{md+6}}{2^{md-n+6} + 2 * (D_{max} * C_{st}) + 2 * (2^{max(d1,s)} + 2^{max(d2,s)} + 2^r)}. \quad (6.24)$$

From this formula we can discuss the range of relative speedup for the parallel computation model.

First, let us consider the case that md is much bigger than $log_2(D_{max} * C_{st})$. In this case, the constant term in formula 6.24 is too small to consider; the formula can be simplified as:

$$rspd \approx \frac{2^{md+6}}{2^{md-n+6} + 2 * (2^{max(d1,s)} + 2^{max(d2,s)} + 2^r)}. \quad (6.25)$$

Then, we can discuss the relative speedup according to the difference of C_{cmp} and C_{cmm}.

1. C_{cmp} is greater than C_{cmm}. If $C_{cmm} = 0$, it is an ideal case or the case of a shared memory model. From formula 6.25, we know that the relative speedup is about 2^n. If $C_{cmm} \neq 0$, one way of getting higher speedup is to increase the number of processors. When increasing the number of processors in the computation, the computation cost at each processors is decreased, but the total communication cost is increased. The number of processors should be increased until C_{cmm} is close to C_{cmp}. Then, the $rspd$ is increased to 2^{n-1}. The problem raised here is if C_{cmm} is too high we may not get enough processors for this computation.

2. C_{cmp} is close to C_{cmm}. From formula 6.25 we know that the relative speedup is about 2^{n-1}. In this case, we can't get much speedup by using more processors. This value is probably the best speedup we can get since the C_{cmm} can never be zero.

3. C_{cmp} is much less than C_{cmm}. More than half of the test results in [DFL92] dropped in to this case. In this case, the relative speedup is determined by the value of communication cost, that is:

$$rspd \approx \frac{2^{md+6}}{2 * (2^{max(d1,s)} + 2^{max(d2,s)} + 2^{r})}. \tag{6.26}$$

The dominant variable affecting C_{cmm} in the above formula is $max(d1, d2, s, r)$. From the formula we can obtain some results for conformal product parallelism. First, the relative speedup has little to do with the number of processors. That is, in this case, increasing the number of processors will not get any impressive improvement in speedup. The test results for the SPI algorithm in [Fou91] showed this point. Second, the communication cost has little to do with the number of processors. This indicates that the large communication cost results mainly from poor factoring results. Third, the relative speedup depends only on the communication cost. Considering the difference between the sequential computation cost and the parallel computation cost, we know that the relative speedup is very low. Therefore, we should avoid this case in factoring for parallelizing probabilistic inference.

Now, let us discuss the influence of factors $d1$, $d2$, s and r in this case. It is obvious that the $max(d1, d2, s, r)$ determines the relative speedup. First, $max(d1, d2, s, r) = r$. The relative speedup is about 2^{md+6-r}. In this case the value of $(md + 6 - r)$ is determined by the number of variables summed over after combination; we denote it rd. Second, $max(d1, d2, s, r) = d1$ (or $d2$). The relative speedup is about $2^{md+6-d1}$. In this case, the relative speedup is determined by the number of variables appearing in both distributions

(remember $md = |S_I \cup S_J|$); we denote it *cmd*. Third, $max(d1, d2, s, r) = s$. This case, called over splitting, can not happen. Because it is only when $C_{cmp} > C_{cmm}$ that we consider splitting more variables than the number of result distribution variables.

From above analyses, we can conclude that the values of *rd* and *cmd* are important factors in considering conformal product parallelism. Besides, the *md* value is important too, because it reflects the computational complexity of the problem itself for a given factoring algorithm. We can see the influence from formula 6.24, when parallel factoring algorithm is different from the sequential algorithm. The factor *cmd* is special in parallel probability computation[20]. Therefore, the key point for designing an efficient parallel probabilistic inference algorithm is to minimize these values.

Second, let us consider the case that *md* is bigger than $log_2(D_{max} * C_{st})$, but not by very much. In this case, $log_2(D_{max} * C_{st})$ will play a role in formula 6.24. Whatever the ratio of C_{cmm} and C_{cmp} is, the speedup is always low. This indicates that conformal product parallelism is not very useful in this case. We should not worry about this case in probabilistic inference in belief networks because we can efficiently solve the problem sequentially.

Two things should be noted in the above discussion. First, the speedup is a relative speedup. If the method of the best sequential algorithm can be used in the parallel case, the relative speedup is also an absolute speedup. Second, the speedup is for one conformal product. Since the largest conformal product dominates the total computation cost in probabilistic inference, the result is valid for the entire computation.

Speedup in evaluation-tree parallelism. From the definition POFP we can see that the entire computation cost for probability computation is, when evaluation

[20] *md* and *rd* are important in sequential probability computation.

tree parallelism is considered, the cost of the highest cost path in the evaluation tree, from the root node to any leaf node. In the ideal case when the cost for each conformal product is the same, evaluation-tree parallelism can reduce the total cost a lot. If there are n conformal products in this case and the tree is balanced, for example, the total cost of evaluation-tree parallelism is $log_2 n$ the time of the single conformal product. However, several constraints limit the role of evaluation tree parallelism. First, the cost for all conformal products is usually not the same. In some cases, one or a few conformal products may dominate the total computation cost; then, the effect of evaluation tree parallelism is very limited. For example, if one conformal product accounts for half of the total cost, the evaluation tree parallelism can only reduce the total cost to $(1/2 + \frac{log_2 n}{n})$ of total cost at most. Second, the complexity of some conformal products is so high (this is the case that requires parallel hardware to handle it) that the number of processors available is exhausted according to the previous discussion. That is, we do not have sufficient processors for more than one conformal product simultaneously. Therefore, we don't expect to obtain a big speedup from evaluation tree parallelism. The experimental test in section 6.7 supports this conclusion.

Efficiency. The efficiency of parallel computation is determined by the speedup and the number of processors used in the computation. If the *rspd* value in the above discussion is an absolute speedup, the efficiency of parallel probability computation can be easily estimated by $rspd/2^n$. The efficiency is proportional to the complexity of the problem when the complexity is big enough according to the analysis for speedup. When the communication cost in the distributed memory model is close to the computation cost, the efficiency of parallelizing probabilistic inference is about 0.5, since the *rspd* is about 2^{n-1} in this case. On the other hand, some conformal products can not be parallelized because of the grainsize; also, those conformal products with the number of variables just a little higher than n have very poor speedup from formula 6.24. Usually, the number of conformal products,

with a small number of variables in a query, is more than half of the total number of conformal products. This reduces the efficiency of parallel probabilistic inference. The experimental test in the section 6 will explore further the relations between the size of a query problem and its efficiency.

Quality of factoring. The value of speedup, in parallel probabilistic inference in belief networks, depends on the quality of a factoring algorithm if the size of the problem is adequate for parallel computation. Since the conformal product parallelism plays a more important role than the evaluation-tree parallelism and a few big conformal products dominate the total computation cost, the values of *md*, *rd* and *cmd* for the biggest conformal products are important in reflecting the quality of a factoring result. Considering that *rd* and *cmd* may change proportionally along with *md*, we can use the ratio $max(d1, d2, r)/md$, denoted *mm*, to describe the quality of a factoring result. The smaller *mm* value, the better the factoring result. A good factoring result doesn't mean a good speedup in parallel computation because a speedup value is also closely related to the problem size. A problem may have a higher *mm* value and a higher speedup compared with another problem. In this case, the problem with higher *mm* is more suitable for parallel computation; for example, a conformal product with higher complexity is more suitable for parallel computation compared with a conformal product with lower complexity. If two problems have the same complexity, a better factoring result will result in a higher speedup. Since we simplify our consideration to parallel probability computation of the biggest conformal product in a query, the *mm* value roughly reflects the quality of a factoring strategy. According to the discussion about C_{cmm} and C_{cmp}, we can also measure the quality of a factoring result by using the ratio: C_{cmm}/C_{cmp}. It is possible to compare the relative speedup with absolute speedup, to judge the quality of a factoring result,t if the best sequential algorithm has a different factoring strategy than a parallel factoring algorithm.

6.6 A heuristic factoring strategy

From the POFP definition and the discussion above, we can present a heuristic factoring strategy for parallelizing probabilistic inference. Because both sequential factoring algorithms and parallel factoring algorithms generate an evaluation tree, a sequential factoring algorithm can be used for parallel probability computation. The following factoring algorithm comes from the sequential factoring algorithm proposed in section 3, chapter 4, including communication cost in step 4.

1. Construct a *factor set* A which contains all factors to be chosen for the next combination. Each factor in set A consists of a set of nodes. Initialize a *combination candidate set* B empty.

2. Add any pairwise combination of factors of the factor set A to B, if the combination is not in set B, except the combination of two factors in which each factor is a marginal node and they have no common child; and compute the $u = (x \cup y)$ and sum(u) of each pair. Where x and y are factors in the set A, sum(u) is the number of nodes in u which can be summed over when the probability computation corresponding to the two factors is carried out.

3. Choose elements from set B such that $C = \{u | u : minimum_B(|u| - sum(u))\}$, here $|u|$ is the size of u excluding observed nodes. If $|C| = 1$, x and y are the factors for the next combination; otherwise, choose elements from C such that $D = \{u | u : maximum_C(|x| + |y|),\ x, y \in u\}$.

4. If $|D| = 1$, x and y are the terms for the next multiplication. Otherwise, choose one element of D which has the minimum $C_{cmm} + C_{cmp}$. If there is more than one element with the same minimum cost, break the tie arbitrarily.

5. Generate a new factor by combining the candidate pair chosen from the above steps and modify the factor set A by deleting two factors of the candidate pair from the factor set and putting the new factor in the set.

6. Delete any pair of set B which has a non-empty intersection with the candidate pair.

7. Repeat step 2 to 5 until only one element is left in the factor set A which is the final combination.

The idea of the algorithm is to minimize the maximum dimensionality of the conformal products and to minimize $max(d1, d2, r)$ value for a query. The problem of minimizing the maximum dimensionality for a query is not exactly POFP. It provides a way of approaching an optimal factoring result by a heuristic strategy. A factoring result with minimal maximum dimensionality for a query may not be optimal. We believe that an optimal factoring result should have minimal maximum dimensionality because the conformal product, with maximum dimensionality, dominates the communication cost and computation cost when the conformal product is big enough that all processors are used. It seems that the problem of minimizing the maximum dimensionality for a query is a sub-problem of POFP since we just try to minimize the largest conformal product; however, the problem is still hard to solve.

6.7 Test results

In this section, we will present the test results for the above heuristic algorithm. We use a set of belief networks generated at random for the test, and we will verify the discussion in section 6.5 and explore further the relation between speedup or efficiency and md for a query in parallelizing probabilistic inference in the next section.

The belief networks for the test were generated using J. Suermondt's random-network generator under the following constraints: for each belief network, the number of nodes was randomly chosen between 50 and 100; the average arcs per node were between 1 and 5; and the number of observations was between 1 and 20.

The query node was chosen at random from the nodes in the network that were not observed. All variables had two values. Since the number of nodes and the average arc per node in a belief network do not represent the graph of a query accurately at times, we will consider the number of conformal products and the maximum dimensionality of conformal products to represent a query in the test. Although, finding the maximum dimensionality of a query depends on a factoring algorithm, the results revealed by the algorithm show us the existence of possible speedup. Also, the maximum dimensionality for a query can be obtained in polynomial time in sequential computation.

Table 4 provides the test results of the 31 test cases which have the maximum dimensionality bigger than the grainsize. The data collected in this table are:

- **cp**, the number of conformal products in an evaluation tree;
- **md**, the maximum dimensionality of conformal products in an evaluation tree;
- **dr**, the value of $max(d1, d2, r)$, where $d1$, $d2$ and r are the size (number of variables) of two input distributions and the result variables respectively in the largest conformal product of a query;
- **mm**, dr/md, the same meaning as in the section 4;
- **cm-cst**, the total communication cost for a query, in microseconds;
- **cp-cst**, the total computation cost for a query, in microseconds;
- **r-spdp**, relative speedup: the ratio of uniprocessor time with multi-processor time for the same evaluation tree;
- **a-spdp**, the absolute speedup, the ratio of uniprocessor time for the best evaluation tree to multi-processor time for this tree;
- **prcs**, the number of processors used;
- **eff**, efficiency of the parallel computation.

The best uniprocessor time for each test case comes from two factoring results. One is from the set-factoring algorithm, which is the best sequential factoring algorithm we know. The other is from the min-max-min algorithm itself; the evaluation tree generated by min-max-min were evaluated by one processor also. We choose a better result of the two as the best sequential result for computing the absolute speedup. The table is sorted according to the *md* value.

Table 5 shows the test results for the algorithm SPI [D'A89], cited from [Fou91][21]. The reason we cite the results of another algorithm is that we want to show the effect of different algorithms for speedup and show the relation of *mm* value with the quality of factoring results.

Table 6 shows the test results of the set-factoring algorithm. The reason we test the sequential algorithm for parallel computation is to compare the results with the results of min-max-min algorithm, and that we want to know the effect of considering communication cost in min-max-min algorithm.

Table 7 presents data of some test cases comparing communication and memory requirements of min-max-min with communication cost heuristic under the BCA model and the dist-net model. The information in this table is as follows.

- **BCA cm**:communication cost for the BCA model.
- **Dist cm**: communication cost for the dist-net model.
- **BCA mem**: memory size used by the BCA model, estimated as the total number of bytes communicated to and from each processor.
- **Dist mem**: memory size used in the dist-net model, estimated as the size of data in the two input distributions plus the result size, for the the largest conformal product.

[21]Tables 5, 6 and 7 are from the test set generated and used in [DFL92]. This test set was expanded for the test in table 4.

#	cp	md	dr	mm	cm-cst	cp-cst	r-spd	a-spd	prcs	eff
1	21	9	9	1.0	1.16+3	3.89+4	1.25	1.25	2	0.625
2	32	10	8	0.8	4.98+3	1.08+5	1.67	1.67	4	0.418
3	35	10	8	0.8	6.56+3	9.42+4	2.3	2.3	4	0.575
4	28	12	9	0.75	8.07+3	8.60+4	3.94	2.03	16	0.127
5	32	13	10	0.77	2.64+4	1.59+5	5.03	5.03	32	0.156
6	37	13	10	0.77	1.71+4	1.49+5	5.89	4.94	32	0.154
7	38	13	11	0.85	1.95+4	1.18+5	6.49	5.33	32	0.167
8	26	14	11	0.79	4.51+4	1.29+5	12.49	12.49	64	0.195
9	42	14	11	0.79	4.14+4	1.74+5	10.29	10.29	64	0.161
10	46	14	11	0.79	2.01+4	1.22+5	9.66	4.44	64	0.069
11	36	15	11	0.73	4.23+4	1.80+5	12.64	11.8	128	0.092
12	32	16	13	0.81	1.15+5	1.68+5	21.57	20.44	256	0.080
13	34	17	13	0.76	1.12+5	1.72+5	36.12	36.12	256	0.141
14	39	17	14	0.82	1.98+5	2.04+5	32.76	32.76	512	0.064
15	43	17	13	0.76	1.05+5	1.93+5	26.75	12.17	512	0.024
16	46	17	13	0.76	9.03+4	1.90+5	30.6	30.6	256	0.120
17	41	19	15	0.79	3.16+5	2.35+5	52.2	39.7	1024	0.039
18	43	19	15	0.73	3.27+5	2.29+5	56.97	56.97	1024	0.056
19	46	19	15	0.79	3.64+5	2.43+5	66.5	66.5	1024	0.065
20	35	21	18	0.86	1.98+6	4.28+5	86.46	47.71	1024	0.047
21	48	22	17	0.77	1.48+6	4.45+5	123.95	53.5	1024	0.052
22	58	22	17	0.77	1.10+6	4.93+5	137.55	137.55	1024	0.134
23	40	26	21	0.81	1.16+7	5.54+6	232.66	91.18	1024	0.089
24	51	26	22	0.85	3.11+7	1.73+7	111.8	111.8	1024	0.109
25	54	29	25	0.86	1.78+8	8.40+7	142.0	142.0	1024	0.139
26	59	29	26	0.90	3.13+8	1.44+8	110.51	110.51	1024	0.107
27	57	31	27	0.87	6.17+8	2.85+8	216.88	216.88	1024	0.212
28	50	35	30	0.87	5.98+9	2.39+9	280.9	72.6	1024	0.071
29	64	35	28	0.80	1.70+9	1.50+9	479.8	479.8	1024	0.469
30	62	42	35	0.83	1.51+11	2.42+11	567.8	528.3	1024	0.516
31	70	45	39	0.87	2.38+12	1.95+12	459.18	459.18	1024	0.484

Table 4. Test results for the min-max-min algorithm.

#	cp	md	dr	mm	cm-cst	cp-cst	r-spdp	a-spdp	prcs	eff
17	41	20	19	0.95	1.77+7	3.99+5	16.4	1.2	1024	.0012
19	46	22	22	1.00	6.30+7	1.38+6	18.7	0.6	1024	.0006
21	48	26	25	0.96	6.20+8	1.32+7	6.3	0.2	1024	.0002
24	51	29	28	0.97	4.08+9	8.39+7	20.0	1.3	1024	.0013
25	54	32	31	0.97	2.27+10	4.99+8	22.0	1.6	1024	.0016
28	50	34	34	1.00	2.04+11	3.68+9	18.2	2.9	1024	.0028
29	64	34	33	0.97	2.48+11	1.37+9	20.1	6.3	1024	.0062
30	62	46	45	0.98	4.52+14	9.80+12	21.8	0.5	1024	.0005

Table 5. Test results for the SPI algorithm.

#	cp	md	dr	mm	cm-cst	cp-cst	r-spdp	a-spdp	prcs	eff
17	41	18	15	0.83	3.39+5	2.76+5	35.5	35.5	1024	.0347
19	46	23	18	0.78	1.64+6	1.34+6	148.3	13.5	1024	.0132
21	48	20	17	0.85	1.04+6	4.76+5	67.9	67.9	1024	.066
24	51	29	25	0.86	2.2+8	4.8+7	159.6	20.0	1024	.0195
25	54	30	24	0.80	1.46+8	7.86+7	249.4	169.8	1024	.1658
28	50	33	29	0.88	2.98+9	6.40+8	168.0	168.0	1024	.1641
29	64	36	31	0.86	9.2+9	3.6+9	256.9	133.3	1024	.130
30	62	42	35	0.83	1.65+11	2.03+11	563.9	563.9	1024	.5507

Table 6. Test results for set-factoring.

#	BCA-cm	Dist-cm	BCA-mem	Dist-mem	memory	m/D-mem
17	3.16+5	1.56+5	4.10+3	2.62+5	2.56+2	1023
19	3.64+5	1.81+5	4.10+3	2.70+5	2.64+2	1023
21	1.48+6	7.39+5	6.15+3	1.31+6	1.28+3	1024
24	3.11+7	1.60+7	1.57+6	2.54+7	1.84+4	1380
25	1.78+8	9.08+7	6.29+6	1.55+8	1.48+5	1047
28	5.98+9	2.99+9	1.68+7	5.40+9	5.28+6	1022
29	1.7+9	8.86+8	2.10+6	1.75+9	1.70+6	1029
30	1.51+11	7.63+10	3.22+9	1.50+11	1.47+8	1020

Table 7. Comparison between the dist-net model and the BCA-CP model.

- **memory**: Memory size used by BCA model ignoring the final conformal product, see discussion.
- **m/D-mem**: Ratio of memory to Dist-mem.

Table 8 provides the results to show evaluation-tree parallelism[22]. In this table, the data are collected from the longest path (or the path that costs the most) in an evaluation tree. The data are sorted according to the *md* value.

- **lcp,** the number of conformal products in the longest path;
- **lcp/cp,** the fractional percentage of the number of conformal products in the longest path to the total number of conformal products;
- **lspd/spd,** the ratio of the relative speedup of evaluation-tree parallelism plus conformal product parallelism with the relative speedup of conformal product parallelism alone;

[22]When considering evaluation-tree parallelism, we have to release the constraint of the total number of processors. That is, we limit the number of processors for a conformal product to 1024, but the total number of processors available is unlimited.

#	cp	md	lcp	lcp/cp	lspd/spd	lcst/cst
1	21	9	7	33.3	1.55	64.5
2	32	10	12	37.5	1.23	81.7
3	35	10	9	25 .7	1.35	74.0
4	28	12	10	35.7	1.54	65.1
5	32	13	9	28.1	1.90	52.7
6	37	13	11	29.7	1.88	53.0
7	38	13	10	26.3	1.85	54.0
8	26	14	8	30.8	1.85	54.1
9	42	14	11	26.2	1.98	50.5
10	46	14	10	21.7	2.01	49.9
11	36	15	7	19.4	2.22	45.0
12	32	16	6	18.8	2.01	49.9
13	34	17	8	23.5	1.75	57.3
14	39	17	8	20.5	1.54	65.0
15	43	17	11	25.6	1.58	63.4
16	46	17	6	13.0	1.80	55.7
17	41	19	7	17.1	1.54	65.2
18	43	19	7	16.3	1.57	63.7
19	46	19	10	21.7	1.40	71.3
20	35	21	9	25.7	1.12	89.4
21	48	22	11	22.9	1.25	80.0
22	58	22	11	19.0	1.23	78.4
23	40	26	9	22.5	1.13	88.8
24	51	26	7	13.7	1.06	94.5
25	54	29	8	14.82	1.06	94.7
26	59	29	9	15.3	1.03	96.6
27	57	31	7	12.3	1.02	97.9
28	50	35	9	18	1.04	96.4
29	64	35	7	10.9	1.03	97.5
30	62	42	10	16.1	1.02	98.5
31	70	45	9	12.9	1.01	99.5

Table 8. Test results of min-max-min for evaluation tree parallelism.

- **lcst/cst,** the fractional percentage of the total cost of evaluation-tree parallelism plus conformal product parallelism to conformal product parallelism alone.

6.8 Discussion

Parallel computation of belief net inferences is feasible. From table 4, we can see that the test results here comply with the analyses in section 6.5. We can get following conclusions from the results. First, good absolute speedup is available for queries with high dimensionality. The poor results (in table 5) in our earlier attempt to parallelize SPI evaluation-trees apparently do not accurately reflect the parallelism available in the underlying computation. Some queries show relatively little speedup, for example from net 2 to net 10. These queries have lower maximum dimensionality, and simply are too small to effectively parallelize (We will see later that the factorings for those queries are as good as the factorings for the other queries in our experiment.). Second, the min-max-min algorithm provides an efficient method for parallel probability computation, although no better than set-factoring. Closeness of **cp-cst** and **cp-cst** values reflects the quality of an algorithm. In table 4, when complexity of a problem is big enough, its computation cost is very close to communication cost, so that the relative speedup is high. While in table 5, the difference of computation cost with communication cost is big; so its relative speedup is low. Third, the maximum dimensionality of conformal products is an important parameter for describing a query, not the physical size of a belief network. From the maximum dimensionality, we may roughly predict the speedup and efficiency for the query. We will discuss this in more detail later.

Characteristics of a Parallelizable Evaluation Tree. When constructing an evaluation tree for sequential computation, only the maximum dimensionality *md* of the tree is important. However, in constructing evaluation trees for parallel com-

putation there are three factors to consider. First, we must consider the maximum dimensionality, as for sequential computation. Second, we must consider the sizes of the two input distributions and the result distribution (for the largest conformal product). As we have discussed before, this will affect the communication costs. Third, we must consider the degree to which the evaluation tree is balanced. This will determine the available evaluation-tree parallelism.

Computation cost for a conformal product is exponential in md. Under optimum conditions (grain size, result variables available for splitting, etc) we can reduce this cost by a factor of n by distributing the computation over n processors. Nonetheless, if an evaluation tree intended for parallel evaluation has significantly higher maximum dimension than the best sequential evaluation tree, we are unlikely to obtain good absolute speedup. From table 4 we can see the larger the dr value, the higher the communication cost. Notice that the dr value for table 5 is always equal to $md-1$. This artifact of the way SPI constructs its evaluation trees explains why the communication cost for SPI is always high. Available speedup under the BCA model is essentially exponential in $md - dr$, since this difference reflects the extra computational burden we can reduce through parallelization.

It seems reasonable to assume that larger nets ought to be more "parallelizable." From the mm value in table 4, we can find that the mm values, for some nets with small speedup, is as low as the values for some larger nets; this indicates that the low speedup obtained can be attributed to the smallness of the problem rather than the quality of the factoring.

Communication/Memory tradeoffs. The differences between the BCA-CP model and the dist-net model are: (1) there is no data distribution needed in the dist-net model; and (2) the dist-net model must store the entire network at each processor. This gives rise to two questions: (1) is there significantly more speedup available under the dist-net model? (2) does the BCA model offer a significantly lower per-processor memory requirement? Our experimental results in table 7 show

that the communication cost for the dist-net model is about half of that for the BCA model. The real impact of communication cost, however, depends on the ratio of computation cost and communication cost. For those queries in which computation time is about 2 to 3 percent of communication time, see table 5, the speedup for dist-net model evaluation would be double that obtained using BCA-CP model evaluation. However, when the computation cost is close to the communication cost, see table 4, the additional speedup using the dist-net model is negligible.

The memory requirements of the alternate evaluation models, in contrast, are quite dramatically different, see table 7. We calculated the memory requirement in the dist-net model as $max_{CPs}(2^{d1} + 2^{d2} + 2^r)$. This value is an estimated optimal result because we ignore all but the largest conformal product in each query. The memory requirement for the BCA model for each processor[23] is calculated as the size of the total data sent to and from a single processor. This value is an upper bound because it assumes each processor must hold all intermediate results. Since the splitting strategy used in the BCA model often limits the parallelism for the last conformal product, which results in an artificially high memory use, we list the memory size ignoring this last conformal product in the column **memory** in table 7. Comparing the values in **BCA mem** and in **memory**, we can see that the memory used in the dist-model is about 1000 times higher than that used by the BCA model. Since we performed all these measurements assuming 1024 available processors, we can therefore conclude that the BCA model is effective in distributing the memory requirement as well as the computational burden.

Why does set factoring perform so well? From table 6 and table 4 we find that the set-factoring algorithm is comparable with the min-max-min algorithm. We did not initially expect that evaluation trees, produced by the sequential version of set-factoring, would perform so well. Why is set-factoring so effective in reducing

[23]There is one processor which has the same size as in the dist-net model, at which results are aggregated. We ignore this processor in this section.

communication cost? Checking the *dr* column in table 6, we can see that the values are not 1s. This means that, in contrast to SPI, set-factoring tends to construct evaluation-trees in which the distributions, being combined, each contain many variables not in the other. Intuitively, we can understand this as a result of the "procrastination" inherent in set-factoring's greedy heuristic. Since set-factoring always seeks the minimum-cost conformal product it can perform next, it tends to produce bushy and balanced trees. But these same trees are exactly the type likely to exhibit large *dr* values, precisely what we need for good speedup.

Why doesn't min-max-min perform even better? We thought, at first, that considering communication cost in the set-factoring algorithm would reduce communication cost significantly so that the speedup would be further improved. Comparing the communication cost of the corresponding test cases in table 6 and table 4, we can find that there is, in fact, little if any improvement. We believe that this indicates that set-factoring, without communication cost, has already found most of the available parallelism, or at least most of that which can be found through simple hill-climbing.

Maximum dimensionality vs. speedup. As for the relations between the maximum dimensionality and speedup, and between the maximum dimensionality and efficiency, these can be seen from figures 13 and 14. Figure 13 is the scatter plot of the *md* value and speedup for each test case in the experimental test. The horizontal axis represents the maximum dimensionality of conformal products for a query and the vertical axis represents the corresponding speedup in logarithm scale. The points in the figure form a curve not a straight line. We can roughly see that the speedup increases with a high rate along with the increase of the *md* value when the *md* value is about less than 18. When the *md* value is bigger than 18, the rate of increasing speedup is lower along with the increase of *md* value. This phenomenon can be explained as follows. When the *md* value is small, not

all processors are used. When md increases, the communication cost increases but the computation cost in each processor has no change because the increased computation is distributed to more processors. For problems of this type, the speedup is affected only by the communication cost until all processors are used. The rate of changing speedup at this period corresponds to the left portion of the curve ($md \leq 18$) with a high slop since the product of the grainsize and the number of processors available is 2^{18}. When all processors are used, increasing md will increase both communication cost and computation cost for each processor and the increased computation cost is distributed to each processor on average. In this case, the increase rate of speedup is lower compared with the cases of $md < 18$, which corresponds to the the right portion of the curve in the figure. Since we use one dominant conformal product to represent the effect of whole conformal products, the curve in the figure is not accurate; but it really reflects the relation of the speedup and the maximum dimensionality.

There is one point in the figure whose absolute speedup is out of the curve. This point with md value 35 has lower speedup than we expected. Its relative speedup is much higher than the absolute speedup. The reason for low absolute speedup is that the sequential algorithm found a factoring result with the md value 33 instead of the 35 in parallel computation[24]. A change of md by 1, between sequential and parallel, should cut absolute speedup by a factor of two. So, the low relative speedup should not be a surprise. This result may reveal that not all networks parallelize well. In order to parallelize well we need lots of non-overlapping variables in the biggest conformal product, without increasing the dimensionality of that conformal product. The min-max-min algorithm couldn't find such a factoring for this network. This case indicates that finding a minimal maximum dimensionality in factoring for parallel computation is still very important, particularly in the case of not enough processors in computation. Also, if conformal product paral-

[24]The speedup for the evaluation tree created by the sequential algorithm is 168.

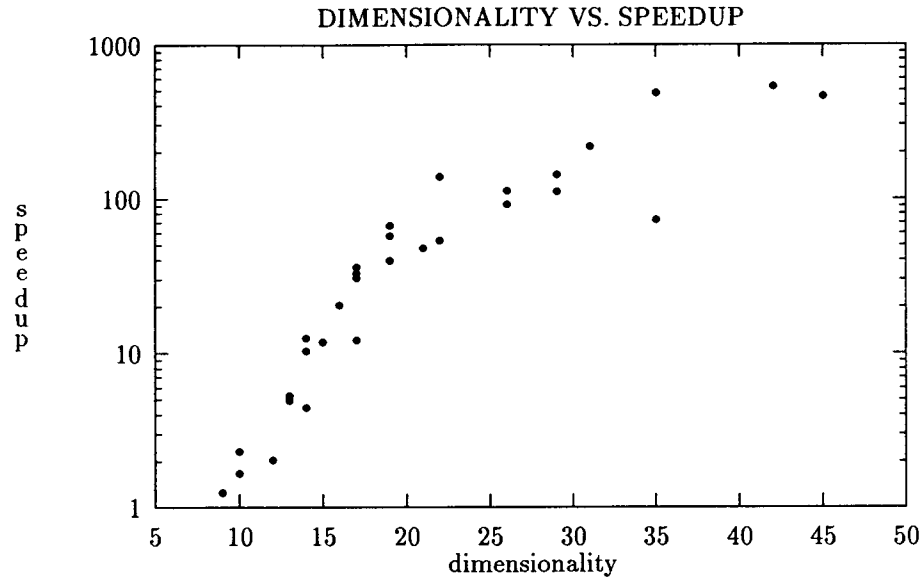


Figure 13. The relation between speedup and maximum dimensionality.

lelism plays a more important role than evaluation-tree parallelism, minimizing the maximum dimensionality is a key point to reduce total computation cost.

Maximum dimensionality vs. efficiency. Figure 14 is the scatter plot for *md* and efficiency for test cases in table 4. The figure reasonably reflects the relation between the maximum dimensionality and the efficiency. When *md* is small, not much parallelism exists, the efficiency is relatively high. When *md* increases to the range of $8 \leq md \leq 18$, both the number of processors used for parallel computation and the speedup increase. The number of processors is increased in exponential proportion with respect to the *md* value. Speedup increases at a lower rate than the number of processors. This is caused by communication cost. Therefore, the efficiency decreased in that period. This can be seen in the left part of the curve in figure 14. When all available processors are used, the increase of *md* results in the increase of speedup only, not in the number of processors. So, the efficiency increases. Because of the communication cost, the efficiency can not get to 1 in the

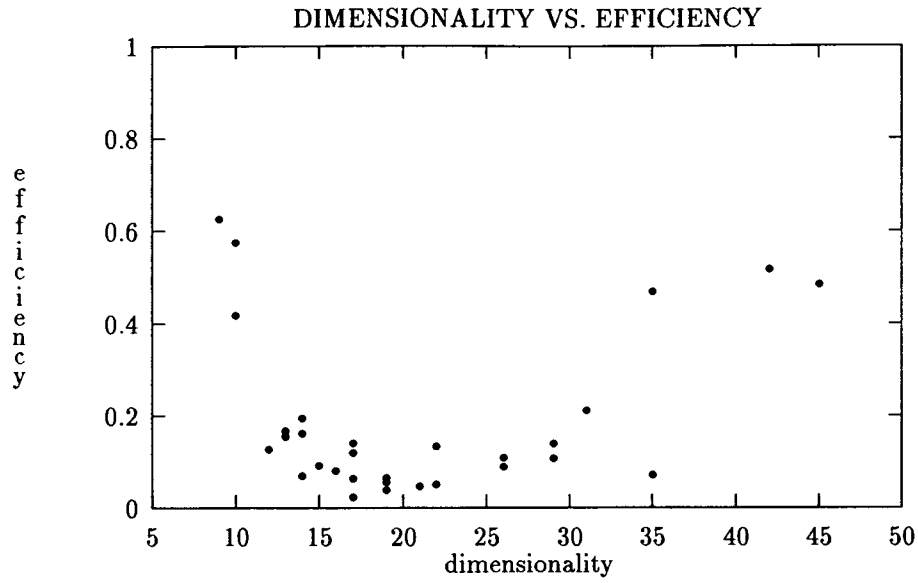


Figure 14. The relation between efficiency and maximum dimensionality.

distributed memory model. The one point with the *md* value 35, having a small efficiency value in the figure, corresponds to the low speedup point in figure 13.

Evaluation tree parallelism. Table 8 tell us that the evaluation-tree parallelism in the min-max-min algorithm is not significant because the ratio of **lspd/spd** is less than 1.5 when *md* is bigger than 19. From column **lcp/cp** we can see that the evaluation tree generated by the min-max-min algorithm is bushy because the fractional percentage of the number of conformal products in the longest path to the total number of conformal products is low, particularly when *md* is bigger than 20. However, from the column **lcst/cst**, we can find that the cost for the longest path is more than half of total cost; that is, the evaluation trees are not bushy in terms of the computation cost. We think that insignificance of evaluation tree parallelism results from the characteristic that one or a few conformal products dominate the total cost. Taking the test case 29 as an example, there are 7 conformal products in the longest path, which is the exact depth of a balanced binary-tree

with 64 leaves; but they consume 97.5% of total cost. The time for one conformal product with dimension 35, with 1024 processors used, is about 10^8 microseconds, which is close to the total computation cost listed in table 4. Furthermore, we may think that the dominance of one or a few conformal products in probabilistic inference is a characteristic of probability computation for large, multiply-connected belief networks in general, because we suspect that an optimal factoring algorithm would reduce the maximum dimensionality significantly for the test cases with big maximum dimensionality. The reasons are that the sequential factoring algorithm, set-factoring, which is very similar to min-max-min, has a better performance with lower maximum dimensionality than any other previously developed algorithms we know [Li90, LD92] , and min-max-min is designed to minimize the maximum dimensionality of a query by combining small factors first and summing over non-queried variables as soon as possible. If the characteristic of dominance is true in probability computation in general, then, the insignificance of evaluation-tree parallelism is true for any parallel probabilistic inference algorithm. We still consider it as an open question.

Figure 15 is the scatter plot for **md** and **lspd/spd** in table 8. There seems a relation between **md** and **lspd/spd**. When $md > 15$, we can see that the ratio **lspd/spd** is decreasing when the *md* is increasing. This means that evaluation tree parallelism, even though small, is inversely proportional to the *md* value. This can be intuitively explained from the dominance of one or a few conformal products. When *md* value is getting higher, the ratio of cost of one or a few conformal products to the total cost is getting higher, so the effect of evaluation tree parallelism is getting smaller. When $md < 15$, the **lspd/spd** is directly proportional to the *md* values. This indicates that the min-max-min algorithm has less chance to get a balanced tree when *md* is small. When *md* is getting bigger, there is more chance of getting a more balanced tree; so, the effect of evaluation tree parallelism is bigger. When *md* is big enough, the biggest conformal product dominates the total

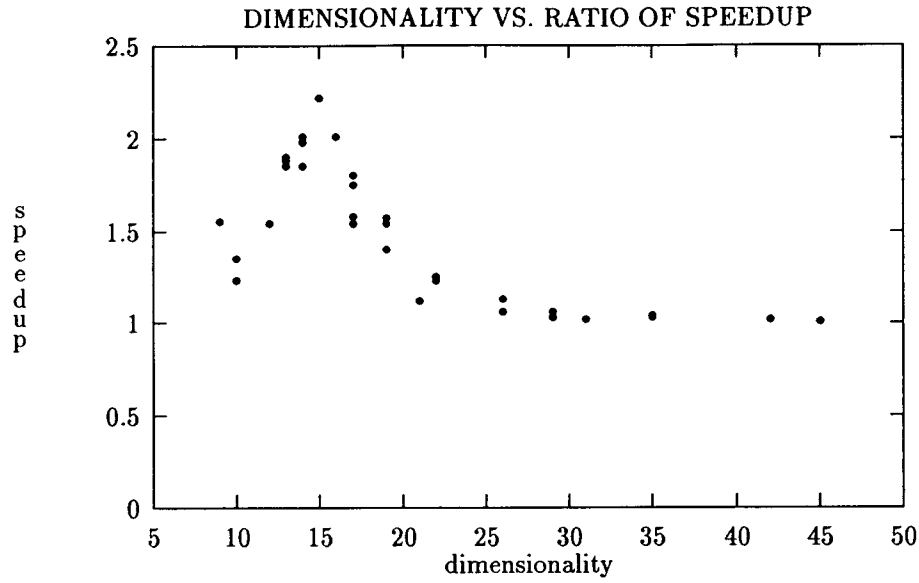


Figure 15. The relation between the ratio of relative speedup and maximum dimensionality.

computation (or the ratio of *md* and *cp* are getting smaller, see table 8), the effect of evaluation-tree parallelism gets smaller.

Other factors affecting speedup There are two other factors that affect available speedup. Sometimes a computation consists of many small conformal products which cannot be parallelized. From the column **para-CP** in table 4 we can see that, on average, parallelizable conformal products are less than half of the total. This usually affects only smaller queries, however. More critical, we believe, is the strategy used for choosing splitting variables. It is possible, as mentioned earlier, to split the variables not in the result. When the input distributions are large, and the result only contains a few variables, our choice to restrict splitting variables to those in the result can significantly limit speedup. One example of this is when the final conformal product has large input distributions, but only one result variable (the query variable). In such cases, better speedup would have been obtained had we split on variables not in the result. This would, however, have required that we

include the cost of aggregating the result (which can be done in log time). However, the memory requirement results indicate this may be worthwhile.

6.9 Conclusion

We have defined a combinatorial optimization problem - an optimal factoring problem for parallelizing probabilistic inference in belief networks. From this definition, we have explored the characteristics of parallelizing probabilistic inference with belief networks in a kind of distributed parallel architecture (hypercube architecture). We have also presented an efficient factoring algorithm for parallel probability computation. The experimental test has verified the conclusion of analysis from the optimal factoring perspective and explored further the relations of the maximum dimensionality of a problem with speedup and efficiency.

Chapter 7

Characteristics of the Factoring Method

7.1 Motivation

As we have mentioned in chapter 2, a number of exact methods have been developed to perform probabilistic inference for marginal, conditional and conjunctive queries in belief networks in recent years[Pea88, Sha86, Sha88, LS88, D'A89, SDD90, JOA90, SKA89, LD92]. It is desirable to compare them and to analyze advantages and disadvantages of each method. There are some reports about the comparisons among some of those methods, such as experimental tests for comparing efficiency of five exact algorithms [Li90], and analytical exploration of the equivalence of some exact methods [SAS91b].

It has been shown that a clustering algorithm is equivalent to all known exact methods except the factoring method described in this paper [SAS91b], which includes some other clustering algorithms [LS88, Pea88, SKA89, FVJO90, JOA90][25], Reduction method [Sha86, Sha88, Sha89, Sha90], Recursive Decomposition method [Coo90b], the Symbolic Probabilistic Inference method [D'A89, SDD90] and Loop-cutset Conditioning method [Pea86, SC90b, HSH90]. The clustering algorithm proposed in [SAS91b] is not a computational improvement over the other methods, except as a unifying framework.

[25] We classify these algorithms in one clustering method.

We have proposed the factoring method for probabilistic inference in belief networks in chapter 3. We believe that considering probabilistic inference in belief networks as a factoring problem captures the essence of the problem. This statement has been supported in several aspects in previous chapters. First, we have shown that efficient probabilistic inference in belief networks is naturally considered as a factoring problem. Second, we have proposed an optimal factoring problem and shown that considering probabilistic inference as a factoring problem provides a framework for finding an efficient inference algorithm. Third, we have shown that finding the l most probable explanations in an arbitrary belief network can be considered as an factoring problem. Fourth, we have shown that it is appropriate to consider parallel probablistic inference as a factoring problem. And finally, we have discussed the issue that probabilistic inference in non-standard belief networks is also a factoring problem. All these problems can be efficiently solved by the factoring method.

In this chapter, we will show further that the factoring method is more general than any other method and has some additional advantages over the others. Approaches for comparing different algorithms or methods are either experimental, analytical or both. However, the approaches for comparing different methods in probabilistic inference can not be complete: experimental comparison can tell us the efficiency of algorithms, but is constrained by the availability of implemented algorithms, and test results may be affected by the implementation of the individual algorithm; analytical comparison may provide some insights into different algorithms, but tell us little about average or expected case performance of heuristics. Every developed algorithm has its own heuristic strategy. Recognizing the difficulties in comparing two methods, we will only discuss some characteristics of the factoring method, instead of comparing it with the others in this section. These characteristics may not be unique to the factoring method itself. However, the integration of them in one method helps to find competitive algorithms for probabilistic inference.

7.2 Characteristics of the factoring method

Quantitative description The quantitative description of the factoring problem in probabilistic inference provides an alternate perspective on the probabilistic inference problem. Quantitatively considering the problem has advantages over graphical consideration. First, the computational results can be easily ranked because the factoring method provides a measure to rank different algorithms. Second, the computation process can be designed for optimality by this quantity instead of by graph structure. Third, some mature techniques used for the optimization problem may be used in the optimal factoring problem.

Dynamic algorithm We think that the factoring method is inherently dynamic. We call an algorithm static if its computation structure (an evaluation tree or clustering tree, etc.) is generated before any observation and query. We know that not all nodes are always relevant to a query, given some observations and the query. Those relevant nodes and their probabilistic relations can form a new graph and the computational structure of the new graph may not be the same as the original graph; so the computation based on the original computation structure may not as efficient as that based on the new one. In the factoring method, all factors are determined after observations and a query (we call it dynamic). The process of computation is query-driven. A query-driven method is usually more efficient than a non-query-driven method since more information is available and considered by a query-driven method.

One advantage of a static algorithm over a dynamic algorithm is that the computation structure needs to be generated once for more than one query. Another advantage of a static algorithm is that it can be compiled off-line for fast response to a query. Since we consider using polynomial factoring heuristics to reduce exponential computation cost, if a dynamic algorithm can efficiently generate an evaluation

tree, then the overhead of the dynamic algorithm will be low in comparison to total inference cost.

Global consideration In factoring method, there are no constraints on combining two factors. Any two nodes can be combined together depending on their relations, on the observations and the query in the belief network, and on the factoring strategy. Considering information globally[26] is a good way to find better factoring results, particularly for multiply-connected belief networks. Even though some of these global optimality issues are NP-hard problems, a heuristic algorithm considering some global information provides impressive results. The set-factoring algorithm is a good example.

Local computations are not necessarily a disadvantage. However, putting some constraints on combination unnecessarily is certainly not an advantage in an algorithm. For example, the triangulation step in some clustering algorithms require clusters be locally determined without considering the influence of any observation and query. We think that the less constrained factoring strategy helps in finding a better factoring result at least in some cases, like the experimental results shown for the set-factoring algorithm in chapter 4.

Intuitive method We think that the factoring method is intuitive and easy to understand and reflects the essence of probabilistic inference problem naturally[27]. A factoring algorithm can be very simple and easily implemented, like the set-factoring algorithm[LD92]. Also the definition of cost function lets us easily evaluate the efficiency of a factoring algorithm. Comparing the process of the set-factoring algorithm and that of a clustering algorithm, the set-factoring is much easier to understand.

[26]The information about two nodes is local if the two nodes are directly connected, otherwise their relation is considered global.

[27]For theoretical proof purpose, we give a not very intuitive definition of the optimal factoring problem.

Optimal results The factoring method provides a way to get optimal factoring results for a query in probabilistic inference. We have developed a dynamic factoring algorithm which finds the optimal factoring results for any belief network[Li91]; but, it takes exponential time with respect to the number of the nodes relevant to a query. Even though the optimal factoring strategy is useless in practice, it is useful in research as an analytical tool to check how close a heuristic algorithm is to an optimal result.

From the characteristics of the factoring method, we can conclude that the factoring method is an efficient way to handle the probabilistic inference in belief networks and generalize the other methods.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the contribution of our research and address some of related problems that we have left open.

8.1 Conclusion

The motivation of our research is to study probabilistic inference and find efficient algorithms to facilitate inference tasks with belief networks. We begin our research by studying and comparing previously developed algorithms, analyzing tasks and representational parameters relevant to the efficiency of the algorithms, and determining the fundamental factors that influence computational cost in querying problems and inference algorithms. In trying to find more efficient algorithms than any developed algorithm, we realized that efficiently solving probabilistic inference in belief networks was a combinatorial optimization problem. In order to apply existing problem solving techniques in the optimization field to the problem, we proposed an optimal factoring problem. Based on the optimal factoring idea, we successfully solved some problems related to probabilistic inference with belief networks. Our findings can be summarized as follows:

1. We proposed that probabilistic inference with belief networks is an optimal factoring problem. The idea of optimal factoring reflects the essence of the

problems of probabilistic inference. We presented an optimal factoring framework for probabilistic inference. This framework fits problems relevant to standard belief networks very well, including parallel probability computation. This framework not only reveals important factors for efficient probabilistic inference but also provides ways of numerically ranking the results. Under this framework we can obtain some results analytically instead of experimentally, which let us obtain some general results more effectively.

2. Under the optimal factoring framework, we classified the computations in probabilistic inference as numeric computation and non-numeric computation. The complexity of numeric computation is exponential with respect to the number of nodes involved in the probability computation, whereas non-numeric heuristics takes polynomial time in the number of nodes. To speed up probabilistic inference is to reduce the time complexity of the numeric computation. The key point in developing an efficient algorithm is to trade numeric computation with non-numeric computation as much as possible. Based on this idea, we have developed some efficient algorithms for solving different problems in probabilistic inference with belief networks.

    - An optimal factoring algorithm for probabilistic inference in singly-connected belief networks. The improvement in reducing computational cost in a singly-connected belief network may not be very important since the computation in singly-connected network is tractable. However, the development of the algorithm reflects the process of applying the results in the optimal factoring problem to probabilistic inference.
    - An efficient probabilistic inference algorithm, set-factoring, for arbitrary belief networks. The algorithm outperforms previously developed algorithms. The algorithm with its efficiency and simplicity gave a good example of using the optimal factoring perspective to generate efficient algorithms.

- The feasibility of parallelizing probabilistic inference. Analysis from the factoring framework and experimental test with a factoring algorithm has shown that parallelizing probabilistic inference in a distributed memory model with hypercube architectures is feasible. Some characteristics of parallelizing probabilistic inference have been explored. And the relations between the complexity of a query problem and speedup and efficiency have been discussed.
- An efficient algorithm for finding the most probable explanations of a belief network. This algorithm applies to arbitrary belief networks, whereas previously developed algorithms only apply to topologically constrained belief networks.
- A linear time algorithm for finding the next most probable explanation of a belief network after finding the first explanation. This algorithm speeds up the process of finding more than one explanation in a belief network.
- An algorithm for finding the most probable explanation for a subset of variables in belief networks. The mechanism for the problem of finding the most probable explanation for a subset of variables in belief networks has been presented.

The above results proved that considering probabilistic inference as an optimal factoring problem really captures the essence of the problem. We can use this idea to any other problems relevant to belief networks.

8.2 Open problems and future work

Many problems are left open throughout the course of our research. We conclude this thesis with a list of such problems.

1. The time complexity of the optimal factoring problem. We have shown that the optimal factoring problem is very similar to the problem of finding the shortest path among n nodes by passing each node exactly once and believed that the optimal factoring problem is NP-hard. However, we haven't proved this statement yet. Revealing the complexity of the optimal factoring problem may not help a lot in finding new algorithms for solving problems in probabilistic inference, but, the problem itself is an interesting topic in the field of combinatorial optimization.

2. Comparing all exact methods of probabilistic inference. We have compared the efficiency of some algorithms for probabilistic inference [Li90] and Shachter [SAS91b] has compared the generality of some of the methods (not from the efficiency point of view). We may need a comparison of all developed methods along different dimensions, such as tasks and so on. The thorough comparison of those methods can help us to understand more about the techniques used in these methods and to find more efficient methods.

3. New algorithms for solving the marginal, conditional and conjunctive queries. We have developed the set-factoring algorithm and some similar algorithms for these queries. The performance of the set-factoring algorithm is better than the performance of the other algorithms tested on arbitrary belief networks. But these experimental tests are limited by the number of nodes and the average arcs per node. We do not know its performance with the belief networks outside the test boundaries. On the other hand, the set-factoring algorithm is a greedy algorithm; we may find a more efficient algorithm with some other heuristics. Some techniques used in finding the shortest path problem and some heuristic search methods may help to find new algorithms.

4. Parallelizing probabilistic inference. We have shown the feasibility of parallelizing probabilistic inference in distributed memory of the hypercube architecture. However, we don't know the results of parallelizing probabilistic

inference in other hardware structures. The results of the algorithm, min-max-min, which explicitly included communication cost, are not as good as we expected, compared with the results of a sequential factoring algorithm for parallel probability computation. Our conclusion was that the set-factoring algorithm had already found most available parallelism. The experiment for the algorithm is limited, we need to test more cases. Also we need to find some new algorithms for parallelizing probabilistic inference.

5. Algorithms for non-standard belief networks.

   Belief networks can be classified into *standard belief networks* and *non-standard belief networks* according to the mechanism for describing the probabilistic relation of a node with its antecedents and the relation among the antecedents. In a standard belief network, the only mechanism available for describing antecedent interactions is the full conditional distribution across all antecedents. This mechanism directly results in exponential cost in both time for probability computation and space for knowledge representation of probability distribution in the number of antecedents. In a non-standard belief network, there exist some other mechanisms for describing the relations between a node with its antecedents and among these antecedents. The discussions in previous chapters are for standard belief networks only.

   The probabilistic knowledge represented by standard belief networks does not cover all interesting probabilistic models for realistic problems. Using non-standard belief networks may be appropriate in some cases and may even simplify knowledge representation and probability computation. A number of restricted interaction models for antecedents have been identified, which have lower space and time complexity than the full conditional distribution across all antecedents. The noisy-or[Pea88, PR87, Hen90], is such an example and has linear cost in both time and space in the number of antecedents. The noisy-or relationship models independent causes of an event.

An algebraic approach, called local expression languages, for representing this kind of knowledge has been proposed in [D'A91]. The language is capable of representing noisy-or and a variety of other special-case interaction models. The probabilistic inference represented by the language for non-standard belief networks can be computed by a standard belief network inference algorithm[D'A89] and this computation retains the space and time advantages of non-standard belief networks. We can show that conditional, marginal or conjunctive query for the non-standard belief networks is also a factoring problem.

We have not provided an algorithm for some tasks relevant to non-standard belief networks. One of these tasks is the probabilistic inference in multi-level noisy-or networks. A query in these networks is a typical problem in non-standard belief networks. The factoring structure for these problems is more complicated than that for standard belief networks.

BIBLIOGRAPHY

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey., 1989.

[CG89a] E. Charniak and R. Goldman. Plan recognition in stories and in life. In *In Proceedings of the Fifth Workshop on Uncertainty in AI*, pages 54–60. AAAI, 1989.

[CG89b] E. Charniak and R. Goldman. A semantics for probabilistic quantifier-free first-order languages with particular applications on story understanding. In *In Proceedings of Eleventh International Joint Conference on AI*, pages 1074–1079. IJCAI, 1989.

[CJ92] E. Charniak and Santos E. Jr. Dynamic MPA calculations for abduction. In *Proceedings, Tenth National Conference on AI*, pages 552–557. AAAI, July 1992.

[Coo90a] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–406, 1990.

[Coo90b] G.F. Cooper. Bayesian belief-network inference using recursive decomposition(KSL-90-05). Technical report, Knowledge Systems Laboratory, Stanford University, 1990.

[CS90] E. Charniak and S.E. Shimony. Probabilistic semantics for cost based abduction. In *Proceedings, Eight National Conference on AI*, pages 106–111. AAAI, August 1990.

[D'A89] B. D'Ambrosio. Symbolic probabilistic inference. Technical report, CS Dept., Oregon State University, 1989.

[D'A90] B. D'Ambrosio. Factoring heuristics in generalized SPI. Technical report, Oregon State Univ. CS dept., 1990.

[D'A91] B. D'Ambrosio. Local expression languages for probabilistic dependence. In *Proceedings of the Seventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 95–102, Palo Alto, July 1991. Morgan Kaufmann, Publishers.

[Daw92] A. P. Dawid. Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, 2:25–36, 1992.

[Dea90] T. Dean. Coping with uncertainty in a control system for navigation and exploration. In *In Proceedings Ninth National Conference on Artificial Intelligence*, pages 1010–1015. AAAI, 1990.

[DFL92] B. D'Ambrosio, T. Fountain, and Z. Li. Parallelizing probabilistic inference-some early explorations. In *Proceedings of the Eighth Annual Conference on Uncertainty in Artificial Intelligence*, Palo Alto, July 1992. Morgan Kaufmann, Publishers.

[Fou91] T. Fountain. Parallelism in the SPI algorithm: An investigation. Technical report, CS Dept., Oregon State University, 1991.

[Fox88] G. Fox *et al. Solving Problems On concurrent Processors.* Prentice Hall, Englewood Cliffs, New Jersey., 1988.

[FVJO90] S. L. Lauritzen F. V. Jensen and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, pages 269–282, 1990.

[Gol90] R. Goldman. A probabilistic approach to language understanding. Technical report, CS-90-34, CS Dept., Brown University, 1990.

[Gol92] J. L. Golmard. A fast algorithm for finding the k most probable states of the world in bayesian networks. *The Thirteeh International Joint Conference on Artificial Intelligence*, Submitted, 1992.

[GVP89] G. Geiger, T. Verma, and J. Pearl. D-separation: from theorems to algorithms. In *Proceedings of the Seventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 118–125. University of Windsor, Windsor, Ontario, 1989.

[HBH89] D. Heckerman, J. Breese, and E. Horvitz. The compilation of decision models. In *Proceedings of the Fifth Conference on Uncertainty in AI*, pages 162–173, August 1989.

[HD90] M. Henrion and M. Druzdel. Qualitative propagation and scenario-based explanation of probabilistic reasoning. In *Proceedings of the Sixth Conference on Uncertainty in AI*, pages 10–20, August 1990.

[Hec89] D. Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proceedings of the Fifth Conference on Uncertainty in AI*, pages 174–181, August 1989.

[Hen90] M. Henrion. Towards efficient probabilistic diagnosis with a very large knowledge-base. In *AAAI Workshop on the Principles of Diagnosis*, 1990.

[HM89] O. Hansson and A. Mayer. Heuristic search as evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty on AI*, pages 152–161, Aug. 1989.

[HSH90] G.F. Cooper H.J. Suermondt and D.E. Heckerman. A combination of cutset conditioning with clique-tree propagation in the pathfinder system. In *Proceedings of the Sixth Conference on Uncertainty in AI*, pages 273–279, July 1990.

[HSME88] J. R. Hobbs, M. Stickel, P. Martine, and D. Edwards. Interpretation as abduction. In *Proceedings of the 26th Annual Meeting of the Association for computation Linguistics*, 1988.

[Hu82] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley, 1982.

[JOA90] F. V. Jensen, K. G. Olesen, and S.K. Andersen. An algebra of bayesian belief universes for knowledge based systems. *Networks*, 20(5):637–659, 1990.

[KP83] J. H. Kim and J. Pearl. A computational model for causal and diagnostic reasoning in inference engines. In *Proceedings of IJCAI-83.* Karlsruhe, FRG, 1983.

[LAB89] T. Levitt, M. Agosta, and T. Binford. Model-based influence diagrams for machine vision. In *Proceedings of the Fifth Workshop on Uncertainty in AI*, pages 233–244, August 1989.

[Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Rinehart and Winston, Holt, 1976.

[LC78] R. E. Larsen and J. L.. Casti. *Principles of Dynamic Programming.* Marcel Dekker, New York, 1978.

[LD92] Z. Li and B. D'Ambrosio. An efficient approach to probabilistic inference in belief nets. In *Proceedings Ninth Canadian Conference on Artificial Intelligence*, pages 121–127, May 1992.

[Li90] Z. Li. Experimental characterization of several algorithms for inference in belief nets. Technical report, Master's thesis, CS Dept., Oregon State University, 1990.

[Li91] Z. Li. Probabilistic inference as a factoring problem. Tech report, CS Dept., Oregon State University, August 1991.

[LS88] S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, B 50, 1988.

[Nea90] R. Neapolitan. *Probabilistic Reasoning in Expert Systems.* John Wiley & Sons, New York, 1990.

[NW88] Georger Numhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-interscience, 1988.

[Pea84] J. Pearl. *Heuristics.* Addison-Wesley, 1984.

[Pea86] J. Pearl. A constraint-propagation approach to probabilistic reasoning. *Uncertainty in Artificial Intelligence*, pages 357–370, 1986.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, Palo Alto, 1988.

[Peo91] Mark A. Peot. Fusion and propagation with multiple observations in belief networks. *Artificial Intelligence*, 48:299–318, 1991.

[PR87] Y. Peng and J. Reggia. A probabilistic causal model for diagnostic problem solving - part 1: Integrating symbolic causal inference with numeric probabilistic inference. *IEEE Trans. on Systems, Man, and Cybernetics: special issue on diagnosis*, SMC-17(2):146–162, 1987.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization Algorithms and Complexity.* Prentice-Hall, 1982.

[RA88] A. Rege and A. Agogino. Topological framework for representing and solving probabilistic inference problems. *Expert Systems*, November:402–414, 1988.

[Ros73] D. J. Rose. *Graph theory and computing.* Academic Press, New York, 1973.

[San89] E. Jr Santos. Cost-based abduction and linear constraint satisfaction. Technical report, Tech Report CS-91-13, Department of Compute Science, Brown University, 1989.

[San91] E. Jr Santos. On the generation of alternative explanations with implications for belief revision. In *Proceedings of the Seventh Conference on Uncertainty in AI*, Aug. 1991.

[SAS91a] Ross D. Shachter, Stig K. Andersen, and Peter Szolovits. The equivalence of exact methods for probabilistic inference on belief networks. Tech report, Dept. of Engineering Economic Systems, Stanford University, 1991.

[SAS91b] Ross D. Shachter, Stig K. Andersen, and Peter Szolovits. The equivalence of exact methods for probabilistic inference on belief networks. *Artificial Intelligence*, Submitted as a research note, October 1991.

[SB89] S. Srinivas and J. Breese. Ideal: Inference diagram evaluation and analysis in lisp. Technical report, Rockwell Palo Alto Lab technical report, May 1989.

[SC88] J. Suermondt and G. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. Technical report, Technical Report, Stanford University, 1988.

[SC90a] S. E. Shimony and E. Charniak. A new algorithm for finding map assignments to belief networks. In *Proceedings of the Sixth Conference on Uncertainty in AI*, Aug. 1990.

[SC90b] H.J. Suermondt and G.F. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *International Journal of Approximate Reasoning*, (4):283–306, 1990.

[SDD90] R. Shachter, B. D'Ambrosio, and B. DelFavero. Symbolic probabilistic inference in belief networks. In *Proceedings Eighth National Conference on AI*, pages 126–131. AAAI, August 1990.

[SFB89] D. Spiegelhalter, R. Franklin, and K. Bull. Assessment criticism and improvement of imprecise subjective probabilities for a medical expert system. In *Proceedings of the Fifth Conference on Uncertainty in AI*, pages 335–342. Association for Uncertainty in Artificail Intelligence, August 1989.

[Sha86] R. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871 – 882, November-December 1986.

[Sha88] R. Shachter. Probabilistic inference and inference diagrams. *Operations Research*, 36(6):589 – 604, July-August 1988.

[Sha89] R. Shachter. Evidence absorption and propagation through evidence reversal. In *Proceedings of the Fifth Workshop on Uncertainty on AI*, pages 303–310, Aug. 1989.

[Sha90] R. Shachter. An ordered examination of influence diagrams. *Operations Research*, 20:535–563, 1990.

[SKA89] K. G. Olesen F. V. Jensen S. K. Andersen. Hugin - a shell for building belief universes for expert systems. In *Proceedings of the 11th International Joint Conference on*

Artificial Intelligence, 1989.

[Sni92] Moshe Sniedovich. *Dynamic programming.* Marcel Dekker, 1992.

[SS90] P. Shenoy and G. Shafer. Axioms for probability and belief-function propagation. In g. Shafer and J. Pearl, editors, *Readings in Uncertain Reasoning*, pages 575–610. Morgan Kaufmann, 1990.

[ST63] N. Sato and W.F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE, PAS*, pages 944–950, 1963.

[Sy92a] Bon K. Sy. Reasoning MPE to multiply-connected belief networks using message passing. In *Proceedings Tenth National Conference on AI*, pages 570–576. AAAI, July 1992.

[Sy92b] Bon K. Sy. A recurrence local computation approach towards ordering composite beliefs in bayesian belief networks. *To appear in the International Journal of Approximate Reasoning*, 1992.

[TY84] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM J. Computing*, 13:566–79, 1984.

[Wen90] W. X. Wen. Optimal decomposition of belief networks. In *Proceedings of the Sixth Workshop on Uncertainty in Artificial Intelligence*, pages 245–256, 1990.

[Wu90] T. Wu. A problem decomposition method for efficient diagnosis and interpretation of multiple disorders. In *Proceedings of 14th Symp. on computer Appl. in Medical Care*, pages 86–92, 1990.