Efficient Inference in Bayes Networks as a Combinatorial Optimization Problem

Zhaoyu Li and Bruce D'Ambrosio
Department of Computer Science,
Oregon State University, Corvallis, Oregon

ABSTRACT

A number of exact algorithms have been developed in recent years to perform probabilistic inference in Bayesian belief networks. The techniques used in these algorithms are closely related to network structures, and some of them are not easy to understand and implement. We consider the problem from the combinatorial optimization point of view and state that efficient probabilistic inference in a belief network is a problem of finding an optimal factoring given a set of probability distributions. From this viewpoint, previously developed algorithms can be seen as alternative factoring strategies. In this paper, we define a combinatorial optimization problem, the optimal factoring problem, and discuss application of this problem in belief networks. We show that optimal factoring provides insight into the key elements of efficient probabilistic inference, and demonstrate simple, easily implemented algorithms with excellent performance.

KEYWORDS: ***belief network, probabilistic inference, combinatorial optimization, optimal factoring, set-factoring, heuristic algorithm***

1. PROBABILISTIC INFERENCE IN BELIEF NETWORKS

Bayesian belief networks provide an intuitive 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 with mutually exclusive and exhaustive values.[1] Arcs and numeric probability distributions describe

Address correspondence to Zhaoyu Li, 1896 Columbia Street, Eugene, OR 97403.

Received July 1992; accepted November 1993.

[1] We will use the terms node and variable interchangeably.

International Journal of Approximate Reasoning 1994; 11:55–81

655 Avenue of the Americas, New York, NY 10010 0888-613X/94/$7.00

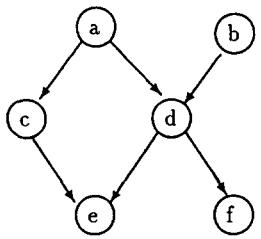
probabilistic relationship between the nodes. A belief network is called singly connected if there is at most one undirected path between any two nodes; otherwise it is called multiply connected. Figure 1 shows a simple multiply connected belief network.

Probabilistic inference in a belief network is the task of computing the probability of a set of variables in the network, given evidence on some subset of the remaining variables. The most common questions we ask in a belief network are: the marginal probability of a node x, the conditional probability of x given y, and the joint probability of a set of variables.

A belief network is a compact representation of a full joint probability distribution over the n domain variables in the network. In particular, the full joint probability distribution can be calculated as follows [14, 16]:

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid \pi_i), \tag{1}$$

where $x_1, \ldots, x_n$ are the n variables in the belief network; π_i is the set of direct predecessors of x_i and $p(x_i \mid \pi_i)$ is the conditional probability for the variable x_i if π_i is not the empty set, and otherwise is the marginal probability of x_i. The product of any two terms of the formula is called a *conformal product*, the number of variables appearing in a conformal product is called its *dimension*, 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.



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.

A number of exact algorithms have been developed in recent years to perform probabilistic inference in belief networks [13, 16, 8, 14, 17, 1, 18, 19, 6, 15, 22]. These algorithms rely either on the original directed graph, on a related directed graph, or on a related undirected graph. For example, the polytree propagation algorithm [14] relies on the original polytree, and the algorithm developed in [8] relies on the related undirected graph. While the graph topology contains all available information for performing inference, much of that information is nonlocal and difficult to extract. We believe that a graph-theoretic perspective from local properties of a network may not be the most effective one from which to develop algorithms for inference in belief networks. Shachter et al. have shown that many of those methods or algorithms derived from those methods are equivalent to a clustering algorithm [20]. This algorithm was proposed not as a computational improvement over the different methods, but rather as a unifying framework in which they can be collectively viewed and combined.

An important characteristic of a belief network is that any conditional, marginal, or conjunctive query in it can be calculated from the full joint probability, and that this is uniquely defined given any network (and a corresponding set of distributions). This can be done by instantiating observed variables in the formula and summing over the variables that are not in the query.[2]

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 [4, 14] provides a way of finding relevant variables to a query in a belief network in polynomial time in the total number of variables.

The variables related to the query correspond to a new belief network (a subgraph of the original network) in which the answer can be obtained by first computing the full joint probability of the new belief network and then summing over the nonqueried variables. The computational cost of inference in belief networks, then, is mainly the cost of computing the full joint probabilities of some subnetwork of the original belief network. This cost can be reduced if variables can be summed over early in the computation, rather than performing all marginalization after computing the full joint probability. The efficiency of probabilistic inference in a belief network, then, depends on finding a *factoring* of the expression for the joint probability over the relevant set of variables, which permits early

[2]A conditional probability $p(X \mid Y)$ can be computed by computing $p(X, Y)$ and $p(Y)$, then using $p(X \mid Y) = p(X, Y)/p(Y)$.

marginalization of variables not in the query. In Section 2, we will formally define an optimal factoring problem and discuss its role in efficient probabilistic inference in belief networks.

2. THE OPTIMAL FACTORING PROBLEM

2.1. An Example

Consider the task of computing the full joint probability of a belief network with n nodes. That task takes at least $2^{n+1} - 4$ multiplications 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 is $(n - 1)/2$ times higher than for the best factoring.

For the case of querying subsets of nodes in belief networks, the variation of the computational cost of different factorings is higher, because the time complexity can vary widely with different factorings. Therefore, the computational cost of probabilistic inference in a belief network depends on the factoring of the conformal product of the distributions for the relevant variables.

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

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

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

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

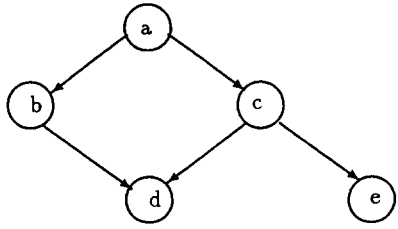


Figure 2. A simple belief network.

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

In this section, 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.

2.2. 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.[3]

DEFINITION 2.1 (FP) *Given*

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

define:

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

$$S_{I \cup J} = S_I \cup S_J - \{v : v \notin S_K \text{ for } K \cap I = \phi,\ K \cap J = \phi, \text{ and } v \notin Q\},$$
$$I, J \subseteq \{1, 2, \ldots, n\}, \quad I \cap J = \phi;$$

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

$$\mu(S_{\{i\}}) = 0 \qquad \text{for} \quad 1 \leq i \leq n,$$
$$\mu(S_{I \cup J}) = \mu(S_I) + \mu(S_J) + 2^{|S_I \cup S_J|}.$$

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

DEFINITION 2.2 (OFP) *The* optimal-factoring *problem is to find a factoring* α *such that* $\mu_\alpha(S_{\{1,2,\ldots,n\}})$ *is minimal.*

In above definitions, Q is a set of target (query) variables; the set $\{v\}$ in the formula $S_{I \cup J}$ is the set of variables which do not appear in the

[3]The mapping between OFP and probabilistic inference in belief networks will be discussed in the next subsection.

remaining subsets of S after removing S_I and S_J and which do not appear in the set Q. The function $\mu_\alpha(S_{I \cup J})$ 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 quantity $2^{[S_I \cup S_J]}$ in the above formula should be replaced with the product of the domain sizes 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 valid ways in the permutation to form all $S_{\{1,2,\ldots,n\}}$.[4]

The OFP generally seems to be a difficult problem. We guess that it is an NP-hard problem, although we have not yet proved this. We can see the similarity between the OFP and the problem of finding the shortest path among n nodes that passes each node exactly once (SPP) [9], which is NP-hard. In the SPP, the problem is to find a permutation of n nodes which results in the shortest path, while in the 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 of the OFP, then 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 [5], the OFP—like the SPP—is the problem of finding a proper permutation of n nodes. The difference between the two problems is that in the SPP edge distances between nodes are static, while in the OFP they are dynamic, that is, they depend on the path taken to the edge.[5]

2.3. Mapping between OFP and Probabilistic Inference

Our interest is in the application of the OFP to probabilistic inference. We can map the problem of finding an optimal evaluation tree for computing the answer to a query in a belief network into an OFP. Given a belief network with m nodes and a set of observations, computing the answer to a query involves identification of a subset of n nodes relevant to the query and computation of the conformal product [19] 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 the OFP: the n nodes

[4]Strictly speaking, there are many apparent duplicates generated in this way. For example, $((ab)c)$ is the same factoring as $(c(ab))$.

[5]An anonymous referee points out that the OFP is very similar to, but not identical to, the "secondary optimization problem" of nonserial dynamic programming (NSDP), which is known to be NP-hard.

with their immediate antecedent nodes are mapped to the n initial subsets; the queried nodes correspond to the variables in the subset Q; $S_{I \cup J}$ denotes the intermediate result for the conformal product of the distributions I and J; and μ gives the number of multiplications needed for this computation. Finding an optimal factoring corresponds to finding an evaluation tree which minimizes the number of multiplications needed for this computation.

We give a simple example to show the mapping between the 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 $\alpha = ((((S_1S_2)S_3)S_4)S_5)$, then $\mu_\alpha(S_{\{1,2,3,4,5\}}) = 28$ (see the example in Section 2.1).

From the OFP point of view, we can view previously developed exact probabilistic inference algorithms as different factoring strategies. However, since these factoring strategies are constrained by the structure of the original graph or a derived graph, it may be hard for them to find optimal factorings.

2.4. Some Results for the OFP

Although the OFP generally is a hard problem, some restricted instances of it 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 $\alpha(S_{\{i_1,\ldots,i_n\}})$ can be obtained in linear time. In this subsection, 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 1 *Given a factoring problem with n variables* $\{1, 2, \ldots, n\}$: $S_1 = \{1\}$, $S_2 = \{2\}, \ldots, S_{n-1} = \{n - 1\}$, $S_n = \{n\}$, *and* $Q = \{1, 2, \ldots, n\}$, *one of the optimal factorings is to combine any* $\text{int}((n + 1)/2)$ *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 two combined marginals, and so on. Remember that the cost function defined in the definition of the OFP is

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

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 $k < m$,

$$\mu(m+1) + \mu(k) > \mu(m) + \mu(k+1). \tag{3}$$

According to the cost function (2), there exist m_1, m_2, k_1, and k_2 which satisfy $m_1 \geq m_2$, $k_1 \geq k_2$, $m_1 + m_2 = m + 1$, and $k_1 + k_2 = k$ such that $\mu(m+1)$ and $\mu(k)$ are both optimal in the left side of (3). If we choose the decomposition for the right side of (3) relevant to m_1, m_2, k_1, and k_2, then to prove (3), we need to prove

$$\mu(m_1) + \mu(m_2) + 2^{m+1} + \mu(k_1) + \mu(k_2) + 2^k \tag{4}$$

is greater than

$$\mu(m_1 - 1) + \mu(m_2) + 2^m + \mu(k_1) + \mu(k_2 + 1) + 2^{k+1}. \tag{5}$$

From (4) and (5) we get

$$\mu(m_1) + 2^{m+1} + \mu(k_2) + 2^k > \mu(m_1 - 1) + 2^m + \mu(k_2 + 1) + 2^{k+1}. \tag{6}$$

Since $2^{m+1} \geq 2^m + 2^{k+1}$, it is sufficient to prove the following inequality instead of (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 sizes k and 1, then the inequality is

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

that is,

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

Thus we should prove the following, since $k \geq k_2 + 1$:

$$\mu(m_1) > \mu(m_1 - 1). \tag{7}$$

The correctness of (7) is obvious for marginals. Thus we have proven (3). From (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 fewer than k marginals and can be combined optimally according to the induction assumption. ■

LEMMA 2 *Given a factoring problem with n variables* $\{1, 2, \ldots, n\}$: $S_1 = \{1\}$, $S_2 = \{2\}, \ldots, S_{n-1} = \{n-1\}$, $S_n = \{1, 2, \ldots, n\}$ *and* $Q = \{n\}$, *the optimal factoring is to combine any* $\text{int}((n+1)/2)$ *single-variable factors according to Lemma* 1, *then to combine the result with the factor* S_n. *The original factoring problem then becomes a new factoring problem with factors* $S_{k_1}, S_{k_2}, \ldots, S_{k_i}, S_{k_{i+1}} = \{k_1, \ldots, k_i, n\}$ *and* $Q = \{n\}$, *where* $i = n - \text{int}((n+1)/2)$, *which has the same form 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 1, is denoted as $M(m)$, then $M(1) = 0$, and $M(m)$, for $m > 0$, can be recursively computed as

$$M(m) = 2^m + M(\text{int}(m/2)) + M(m - \text{int}(m/2)). \tag{8}$$

There is a combination order for $S_1, S_2, \ldots, S_n$ and $m = \text{int}(n/2)$ such that the number of multiplications for combining

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

is

$$2^n + (2^2 + \cdots + 2^m) + 2^{n-m} + (2^2 + \cdots + 2^{n-m-1}). \tag{9}$$

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

$$2^2 + 2^3 + \cdots + 2^{n-1} = 2^n - 4. \tag{10}$$

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

$$F(n) = 2^n + M(m) + F(n - m). \tag{11}$$

Then, proving the combination order for combining $S_1, \ldots, S_n$ is equivalent to proving the following inequality: Given s and t, with $s \neq t$ and t the number chosen as in the lemma, then

$$2^n + M(s) + F(n - s) > 2^n + M(t) + F(n - t). \tag{12}$$

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 inequality in s and t:

$$M(t - j) + F(n - t + j) > M(t) + F(n - t). \tag{13}$$

From (8) and (9) we know that the dominant terms in the left side of (13) are $2^{t+j} + 2^{t-j} + \cdots$. The rest of the terms are much smaller. The dominant terms in the right side of the formula are $2^t + 2^t + \cdots$, and the rest of the terms are again much smaller. According to (10), we know that (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

$$M(t + 1) + F(n - t - 1) > M(t) + F(n - t). \tag{14}$$

From (8) and (9) we know that the dominant terms in the left side of (14) are $2^{t+j} + 2^{t-j} + \cdots$, and the rest of the terms are much smaller. Similarly, the dominant terms on the right side are $2^t + 2^t + \cdots$, and the rest of the terms can be ignored in comparison. This tells us that (14) is correct for $s = t + j$ for $1 \leq j \leq t$.

Similarly we can prove (12) in the case $n = 2t - 1$ for $s < t$ or $s > t$. For $s > t$ the proof is similar to the above proof. For $s < t$, if we substitute s in (12) with $s = t - 1$, the two sides are equal. Thus if we use $s < t - 1$ instead of $s = t - 1$, (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 (12), we combine t marginals first. t is determined as above. Then we 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 $S_1, S_2, \ldots, S_n$ is proved. ■

LEMMA 3 *Given a factoring problem with n variables* $\{1, 2, \ldots, n\}$: $S_1 = \{1\}$, $S_2 = \{2\}, \ldots, S_n = \{n\}$, $S_{n+1} = \{1, 2, \ldots, n\}$, *and* $Q = \{n\}$, *the optimal factoring is to combine* $S_1, \ldots, S_{n-1}$ *with* S_{n+1} *first, then combine the result with* S_n. *The order of combining* $S_1, \ldots, S_{n-1}$ *with* S_{n+1} *is given in Lemma* 2.

Proof We can see that to combine $S_1, \ldots, S_{n-1}$ with S_{n+1} is the same factoring problem described in Lemma 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 combination given in Lemma 3 is optimal. ■

LEMMA 4 *Given a factoring problem with* $n + k - 1$ *sets on n variables* $\{1, 2, \ldots, n\}$: $S_2 = \{2\}, \ldots, S_{n-1} = \{n - 1\}$, $S_n = \{1, 2, \ldots, n\}$, k $\{1\}$*'s (we may denote them as* $S_{1,1} = \{1\}, \ldots, S_{1,k} = \{1\}$*), 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 Lemma* 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 Lemma 3. Therefore, the factoring in the lemma is optimal. ■

LEMMA 5 *Given a factoring problem with n variables* $\{1, 2, \ldots, n\}$: $S_1 = \{1\}$, $S_2 = \{1, 2\}$, $S_3 = \{2, 3\}, \ldots, S_n = \{n - 1, n\}$, *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 [11, 12, 5]. It is related to branch-and-bound techniques in the sense that it performs an

intelligent enumeration of all feasible points of a problem. The idea is to work backwards from the last decisions to the earlier ones. Using the dynamic programming approach to the OFP, we start backwards from an assumed optimal result. According to the "principle of optimality," any subcombination of n factors must be an optimum itself, and all possible subcombinations may be used in the final optimal result. We keep all computed optimal subcombinations, 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 total number of 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 will be $O(n^2 \times 2^n)$ 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 tradeoff 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 extending the heuristic methods used for SPP to OFP; for example, we may use the "nearest neighbor" strategy in OFP. We will not further explore that possibility in this paper.

3. OPTIMAL FACTORING FOR SINGLY CONNECTED BELIEF NETWORKS

From Section 2 we know that finding 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 polytree) belief networks. In this section, we will present the algorithm. The optimal factoring algorithm is based on the lemmas in Section 2.

The meaning of Lemma 2 to Lemma 5 of Section 2 for networks can be shown with the help of very simple belief networks. Lemma 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 for computing the marginal probability of the nth variable. Lemma 3 refers to a similar graph, where the query is for the conditional probability of $p(n \mid n + 1)$ and node $n + 1$ is a child of node n and is observed. Lemma 4 describes a more general case shown in Figure 3(c). A ample query for the graph is $p(1 \mid 2, 3, \ldots, n)$, where node i $(i > 1)$ is observed. Lemma 5 refers to a belief network with chain structure [see Figure 3(b)] in which the marginal probability of node n or the marginal probability of node 1, given observation of node n, is queried. The lemma tells us that the cost of combining two nonmarginal nodes which are not directly connected is always greater than the cost of combining two nodes

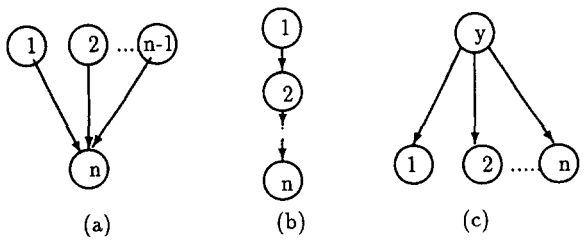


Figure 3. Different cases of query is polytrees.

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 the group *head*; a marginal node is the only node in a group and is the group head.

THEOREM 1 *There exists a linear-time algorithm to generate an optimal factoring for querying the marginal probability of a node in a polytree.*

Proof Based on the factoring strategies in Lemma 2 to Lemma 5, we can construct an optimal factoring strategy for a polytree. Given some observed nodes and a queried node in a polytree, the nodes relevant to the query still form a polytree. The nodes that are the antecedents of the queried node in the original polytree are in the reduced polytree, and the descendants of those antecedents must be observed nodes or antecedents of some observed nodes. A queried node divides all nodes of the reduced polytree into two parts, successor nodes and antecedent nodes. 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 for the group head should be computed first.
2. *Top-down.* In computing the marginal probability of a group head, if the head has any children, then the groups with each child as the head should be computed first with the head of the first group as the target variable.

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. 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 Lemma 2 or 3 to compute the target variable of the group. If some computed group has the form in Figure 3(c), then apply Lemma 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 in the factoring strategy have forms given in Lemma 2 or can be converted

to one of those forms. If the group with the queried node as a head cannot be computed in one of those 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, since each root node is either a marginal node, an observed node, or the queried node in the formed polytree. 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 nonhead nodes, and they either take on form 1 or need top-down operation. If some of them are in form 1, they can be computed again, and so on for the other groups.

The groups generated from top-down operation either are in form 2 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 as described above. Those groups generated by repeatedly using top-down operation must be in form 2, because a leaf in the reduced polytree is an observed node. By applying Lemma 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 4. The group having a returned value then takes on form 2. Notice that we take the group in form 1 here because the group with a returned value can be computed in Lemma 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 5. This can easily be shown by induction on the number of groups in the polytree. 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 the polytree propagation algorithm [7, 14] and the revised polytree algorithm [15], provide an optimal factoring among groups for computing probabilities; but their propagation strategies do not provide any factoring strategy within a group.

4. FACTORING IN MULTIPLY-CONNECTED BELIEF NETWORKS

We doubt if there exists a polynomial-time optimal-factoring algorithm for an arbitrary belief network, because we believe that the OFP is an NP-hard problem. In this section we will present an efficient heuristic

factoring algorithm. After presenting the algorithm, we will discuss some considerations in designing a factoring strategy for multiply connected belief networks.

4.1. A Heuristic Factoring Algorithm for Arbitrary Belief Networks

There are three points for reducing the computational cost of probabilistic inference in belief networks: minimizing the maximum dimensionality of a query, avoiding unnecessary computation, and reducing repeated computation.

The problem of minimizing the maximum dimensionality for a query is not exactly the OFP. A factoring with minimized dimensionality for a query may be suboptimal in total number of multiplications, while an optimal-factoring result will usually have minimal dimensionality. Nonetheless, minimization of dimensionality is a good approximation of the OFP for most queries, and is the intuition behind the heuristic algorithm we present in the following.

4.1.1. 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 $\{x_1, x_2, \ldots, x_n\}$ connected by arcs, the general form of a query is $P(X_J \mid X_K, X_E)$, 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. $P(X_J \mid X_K, X_E)$ can be computed from $P(X_J, X_K \mid X_E)$. For simplicity, we will only consider the case $P(X_J \mid X_E)$ in the algorithm. This ignores several potential simplifications noted in [19], but simplifies the presentation.

Given a query $P(X_J \mid X_E)$ 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 [4]. Once we have obtained the nodes needed for the query, we have all the factors to be combined. In accordance with Definition 2.1, we have n subsets of n nodes and the 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 (initially all the relevant network distributions). Each factor in A is represented as a set of variables. Initialize a *combination candidate set* B empty.
2. Add all pairwise combinations of factors of the factor set A to B which are not already in B, except those combinations in which each factor is a marginal factor and they have no common child; and compute $u = x \cup y$ and sum(u) of each pair, where x and y are

factors in the set A, and sum(u) is the number of variables in u which can be summed over when the conformal product corresponding to combining the two factors is carried out.

3. Choose elements from the set B such that $C = \{u \mid \min_B(|u| - \text{sum}(u))\}$, here $|u|$ is the size of u excluding observed nodes. If $|C| = 1$, then x and y are the factors for the next combination; otherwise, choose elements from C such that $D = \{u \mid \max_C(|x| + |y|), x, y \in u\}$. If $|D| = 1$, x and y are the terms for the next multiplication; otherwise, choose any member of D.
4. Generate a new factor by combining the pair chosen in the above steps. Modify the factor set A by deleting the two factors of the chosen pair from the factor set and adding the new factor in the set.
5. Delete any pair in the set B which has nonempty intersections with the candidate pair.
6. Repeat steps 2 to 5 until only one element is left in the factor set A, which is the final result.

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 two possible values of each variable. The nodes relevant to the query are {1, 2, 3, 4}. We use the set-factoring algorithm to combine the distributions:

Loop 1. The 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. $p(2 \mid 1) \times p(1)$, 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.

Loop 2. The 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.

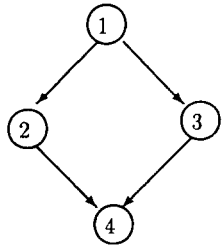


Figure 4. A simple belief network.

Loop 3. The 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

$$p(4) = \sum_{2,3}\left(p(4 \mid 2,3)\left(\sum_{1}(p(3 \mid 1)(p(2 \mid 1)p(1)))\right)\right).$$

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 values of all nodes is the same. If the numbers of values of the nodes in a belief network are different, we can consider the product of the numbers 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 combining any two factors, we check the caching table to see if there is a cached result for the combination. If there is, we can use it 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 for multiple queries, and in fact makes this approach as efficient as clique-tree approaches in computing all marginals.

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.[6] 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 the dimensionality. In step 3, we choose the pairs which have the lowest resulting dimensionality as candidates, since the best result of the current combination may need fewer 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 primarily a function of the number of nodes related to the current query. Step 1 is linear in the

[6]The number of multiplications should be $2^{|x \cup y|}$.

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 are a total of $[n(n-1)/2] + \Sigma_k(n-k) = n^2 - 3n + 3$ 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 numbers of pairs left in the sets B and C respectively; it is at most $O(n^2)$, including $n-1$ loops needed for the two steps. The modification of the factor set in step 4 is linear in the number of factors; it has at most n elements. Deleting some elements from the set B in step 5 is linear in the number of 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.

4.1.2. 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 [10] with the implementation of the IDEAL system [21] for conditioning, clustering, and reduction algorithms and with the implementation of SPI [1]. Since SPI had equal or better performance in every case in that study, in this section we experimentally examine set factoring with SPI only.

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[7] (ranging from 1 to the number of nodes in 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 the 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 that we want to compare the results of set factoring with those of an optimal algorithm, which is limited to running small belief networks because of time complexity.[8] Table 1 shows the characteristics of

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

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

Table 1. Ten Small Test Cases and the Test Results by Algorithms[a]

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

[a]The generalized SPI, the set-factoring, and the optimal algorithm.

the 10 test cases and the computational results of different algorithms measured in the number of multiplications. The data collected in this table are the following:

- **net**, the index of test cases;
- **node**, the number of nodes in each belief network;
- **arc / n**, the average number of 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 [19];
- **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 and the test results. Columns 2 to 4 show the number of nodes, the number of observations, and the number of queries for each test case. 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 give optimal results for some test cases.

The third set of test cases is that used in testing SPI and generalized SPI [1, 2, 19]. They are randomly generated from 1.0 to 5.0 arcs per node and 10 to 30 nodes. In Table 3, **n** is the number of nodes and **a** the number of arcs in each belief network; **o** and **q** are the numbers 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

Table 2. Tree Structured Test Cases and Test Results by Algorithms[a]

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

[a]The generalized SPI, the set-factoring, and the optimal algorithm.

Table 3. The Experimental Results of 21 Test Cases between SPI and Set Factoring

No.	n	a	o	q	SPI	set-f	SPI-cach	set-cach
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

is used for comparison. **SPI-cach** and **set-cache** show the results with intermediate-result caching for both algorithms.[9]

From the above experimental results we see that the factoring strategy of set factoring gives better factoring results than those of SPI in every case, particularly when the belief network is large. The number of multiplications in set factoring is about half of that in SPI on average. Set factoring is more consistent with respect to tasks and different kinds of belief networks. As shown in Table 3, set factoring is better than SPI with caching for a large belief network: take network 16 as an example. Since the dimension in a factor will become large after some combinations, any bad combination order will cause many more multiplications than a good one 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 complexity is at most $O(n^3)$ in the number of nodes concerned in the current query; in SPI it is at most $O(n^3)$ in the number of nodes of the belief network. The actual time cost for symbolic reasoning in both algorithms is trivial compared to probability computation.

4.2. 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 the 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 the 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 one step in advance 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-struc-

[9]The asterisk denotes that the algorithm is too slow to run the test case.

tured belief networks: the factoring information for a 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. A simple extension would be to look further ahead, for example to choose triplets or quadruplets. We have not tried this idea.

Since the last several combinations in set factoring usually have large dimensionality, combinations of them are critical in getting nearly optimal results. Considering this, we combined the set-factoring and the optimal algorithm 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 eight factors, the combined algorithm should run efficiently as well. The results of the combined algorithm are better than those of the set-factoring algorithm, particularly for large belief networks.[10] This led us to think of another factoring strategy 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 on 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 depend not only on an optimal factoring strategy, but also on a good caching method for some tasks and some belief networks. There is a tradeoff 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.

We have also studied the opportunities for parallelism in belief-network inference. Set factoring has shown good factoring results for parallelizing probabilistic inference [3].

4.3. Features for Efficient Probabilistic Inference in Belief Networks

In this subsection we discuss some influences on the efficiency of algorithms for probabilistic inference.

4.3.1. FACTORING VS. NUMERIC COMPUTATION We refer to the computation of conformal products as *numeric computation.* We find that the numeric computation in probabilistic inference is exponential in the number of variables relevant to the computation, while factoring heuristics are

[10] Take network 16 in Table 3 as an example: the number of multiplications needed by the combined algorithm is about 75% of that by set factoring.

typically polynomial with respect to the number of variables related to the query [10]. The factoring computation can be very small if we simply randomly combine the distributions for a query and sum over those variables not queried. However, the total computational cost (factoring plus conformal products) could be quite high in that case. The factoring computation can be very expensive if we want to minimize the numeric computation. It is important to realize that the critical task for factoring 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, one should spend a lot of time searching for low maximum dimensionality if the maximum dimensionality is large, since the payoff from such a search is potentially very high. In the case where the number of nodes relevant to a query is large but the maximum dimensionality is relatively low, the cost of factoring should be limited to a low-degree polynomial. It should be clear that the maximum dimensionality of a query given an algorithm reflects the real computational complexity of the query in a belief network for the 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 much interested in finding an algorithm which performs probabilistic inference in a belief network with the minimal maximum dimensionality.

4.3.2. 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-subsection, we will discuss the advantages and disadvantages in static and dynamic factoring strategies 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 [1], 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. A disadvantage 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 are very different for a given belief network, the constraints may exclude optimal factorings for some queries.

Dynamic factoring is performed at query time, 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. 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 may be less effective.

One possible difference between static factoring and dynamic factoring is the reusability 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 the belief network and the tasks to be carried out, as the test results indicated in [1, 19]. Some tasks favor caching: for example, 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 a long chain will provide many opportunities for caching when the queried nodes are all in the chain.

An experimental test has been performed for examining the effects of caching between the SPI algorithm [1], 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 Section 4.1). The experiment showed that the effect of caching for the set-factoring algorithm is significant and is comparable to that for the static factoring algorithm (SPI). These results indicate caching is useful in dynamic factoring algorithms.

5. CONCLUSIONS

In this paper we have presented a combinatorial optimization problem, optimal factoring. We have proposed that efficient probabilistic inference in a belief network can be considered as an optimal factoring problem. We believe that it is a proper way to study the problem. From this point of view, finding an efficient exact probabilistic inference algorithm means finding an optimal factoring algorithm. Unfortunately, finding an optimal factoring in general is a hard problem. Currently developed algorithms rely on structural properties of the graph to guide factoring. However, it is not clear this is the most direct way to find efficient factorings. We presented a heuristic factoring algorithm for multiply connected networks which makes no reference to graphical structure and yet outperforms current graph-based algorithms.

References

1. D'Ambrosio, B., Symbolic probabilistic inference, Tech. Report, CS Dept., Oregon State Univ., 1989.

2. D'Ambrosio, B., Factoring heuristics in generalized SPI, Tech. Report, CS Dept., Oregon State Univ., 1990.

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

4. Geiger, G., Verma, T., and Pearl, J., D-separation: From theorems to algorithms, in *Proceedings of the Seventh Annual Conference on Uncertainty in Artificial Intelligence*, Univ. of Windsor, Windsor, Ontario, 118–125, 1989.

5. Hu, T. C., *Combinatorial Algorithms*, Addison-Wesley, 1982.

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

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

8. Lauritzen, S., and Spiegelhalter, D., Local computations with probabilities on graphical structures and their application to expert systems, *J. Roy. Statist. Soc. Ser. B* 50, 1988.

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

10. Li, Z., Experimental characterization of several algorithms for inference in belief nets, Tech. Report, Master's Thesis, CS Dept., Oregon State Univ., 1990.

11. Numhauser, Georger, and Wolsey, Laurence A., *Integer and Combinatorial Optimization*, Wiley-Interscience, 1988.

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

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

14. Pearl, J., *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufmann, Palo Alto, 1988.

15. Peot, Mark A., and Shachter, R., Fusion and propagation with multiple observations in belief networks, *Artificial Intelligence* 48, 299–318, 1991.

16. Shachter, R., Evaluating influence diagrams, *Oper. Res.* 34(6), 871–882, Nov.–Dec. 1986.

17. Shachter, R., Probabilistic inference and inference diagrams, *Oper. Res.* 36(6), 589–604, July–Aug. 1988.

18. Shachter, R., Evidence absorption and propagation through evidence reversal, in *Proceedings of the Fifth Workshop on Uncertainty on AI*, 303–310, Aug. 1989.

19. Shachter, R., D'Ambrosio, B., and DelFavero, B., Symbolic probabilistic inference in belief networks, in *Proceedings Eighth National Conference on AI*, AAAI, 126–131, Aug. 1990.

20. Shachter, Ross D., Andersen, Stig K., and Szolovits, Peter, The equivalence of exact methods for probabilistic inference on belief networks, Tech. Report, Dept. of Engineering Economic Systems, Stanford Univ., 1991.

21. Srinivas, S., and Breese, J., Ideal: Inference diagram evaluation and analysis in Lisp, Tech. Report, Rockwell Palo Alto Lab., May 1989.

22. Zhang, L., and Poole, D., Sidestepping the triangulation problem in bayesian net computations, in *Proceedings of the Eighth Annual Conference on Uncertainty in Artificial Intelligence*, Palo Alto, July 1992, Morgan Kaufmann.