Inference in Bayesian Networks: The Role of Context-Specific Independence

*Nevin L. Zhang**

Technical Report HKUST-CS98-09

*Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

Three kinds of independence are of interest in the context of Bayesian networks, namely conditional independence, independence of causal influence, and context-specific independence. It is well-known that conditional independence enables one to factorize a joint probability into a list of conditional probabilities and thereby renders inference feasible. It has recently been shown that independence of causal influence leads to further factorizations of some of the conditional probabilities and consequently makes inference faster. This paper studies context-specific independence. We show that context-specific independence can be used to further decompose some of the conditional probabilities. We present an inference algorithm that takes advantage of the decompositions and provide, for the first time, empirical evidence that demonstrates the computational benefits of exploiting context-specific independence.

1 Introduction

The probabilistic approach to reasoning under uncertainty has grown to prominence in the last fifteen years or so. It models a problem domain using a set of random variables, represents uncertain knowledge and beliefs as probabilistic assertions, and makes inference, i.e. computes posterior probabilities, according to the laws of probability.

In principle, arbitrary queries about posterior probabilities can be answered if there is a joint probability over all variables. The amount of numbers it takes to specify a joint probability is exponential in the number of variables. For this reason, probabilistic inference was thought to be infeasible until the introduction of Bayesian networks (BNs) ¹ (Pearl 1988, Howard and Matheson 1984). Making use of conditional independence, a BN factorizes a joint a probability into a list of conditional probabilities: the joint probability can be recovered by multiplying the conditional probabilities. The factorization renders knowledge acquisition and inference feasible because each of the conditional probabilities involves only a fraction of the variables.

Recently, there is much interest in exploiting structures in the conditional probabilities. One major source of structures is independence of causal influence (ICI) ² (Heckerman 1993). ICI refers to situations where multiple causes influence a common effect independently. Olesen *et al* (1993), Heckerman (1993), and Heckerman and Breese (1994) point out that ICI can be used to simplify the topology of a BN and hence speed up inference. A more efficient approach is developed by Zhang and Poole (1994, 1996). The approach is based on the observation that ICI allows one to further factorize a conditional probability into a list functions: the conditional probability can be recovered by combining the functions using a convolution-like operator. While the conditional probability might involve a number of variables, each of the functions involve only two variables. Knowledge acquisition and inference are therefore made easier.

This paper is concerned with another major source of structures, namely context-specific independence (CSI) (Boutilier *et al* 1996). CSI refers to conditional independencies that hold only in specific contexts. We observe that CSI allows one to further decompose a conditional probability into a list of partial functions: the conditional probability can be recovered by taking the “union” of the partial functions. The partial functions require fewer numbers to specify than the conditional probability and hence knowledge acquisition and inference becomes easier. We develop an inference algorithm that takes advantage of the further decompositions and provide, for the first time, empirical evidence that demonstrates the computational benefits of exploiting CSI.

The rest of this paper is organized as follows. We will start with a brief review of BN (Section 2) and a brief review of CSI (Section 3). The notions of partial function and decomposition will be formally defined in Section 4. In Section 5, we will present a high level inference algorithm that works with the decompositions induced by CSI. The algorithm will be refined in Sections 6 to 8, alongside a detailed account of the important

¹Also known as probabilistic influence diagrams and belief networks.

²Also known as causal independence.

issue of preserving structure during inference. Empirical results will be reported in Section 9, related work discussed in Section 10, and conclusions provided in Section 11.

2 Bayesian networks

A *Bayesian network* (BN) is an annotated directed acyclic graph, where each node represents a random variable and is attached with the conditional probability of the node given its parents. In addition to the explicitly represented conditional probabilities, a BN also implicitly represents conditional independence assertions. Let $x_1, x_2, \dots, x_n$ be an enumeration of all the nodes in a BN such that each node appears before its children, and let π_{x_i} be the set of parents of a node x_i . The following assertions are implicitly represented:

For $i=1, 2, \dots, n$, x_i is conditionally independent of variables in $\{x_1, x_2, \dots, x_{i-1}\} \setminus \pi_{x_i}$ given variables in π_{x_i} .

The conditional independence assertions and the conditional probabilities together entail a joint probability over all variables. As a matter of fact, by the chain rule, we have

$$P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n P(x_i | x_1, x_2, \dots, x_{i-1}) = \prod_{i=1}^n P(x_i | \pi_{x_i}), \quad (1)$$

where the second equation is true because of the conditional independence assertions. The conditional probabilities $P(x_i | \pi_{x_i})$ are given in the specification of the BN. Because of (1), we say that the BN *factorizes* the joint probability $P(x_1, x_2, \dots, x_n)$ into conditional probabilities $p(x_1 | \pi_{x_1})$, $p(x_2 | \pi_{x_2})$, $\dots$, and $p(x_n | \pi_{x_n})$ and that the conditional probabilities constitute a *multiplicative factorization* of the joint probability. The BN in Figure 1, for instance, gives us the following multiplicative factorization of $P(x_1, x_2, \dots, x_5)$:

$$P(x_1), P(x_2), P(x_3 | x_1, x_2), P(x_4 | x_1, x_2), P(x_5 | x_1, x_3, x_4). \quad (2)$$

We will use this network as a running example through out the paper.

3 Context-specific independence

Let C be a set of variables. A *context* on C is an assignment of one value to each variable in C . We denote a context by $C=\gamma$, where γ is a set of values of variables in C . Two contexts are *incompatible* if there exists a variable that is assigned different values in the contexts. They are *compatible* otherwise.

This following definition of context-specific independence (CSI) is due to Boutilier *et al* (1996). Let X, Y, Z , and C be four disjoint sets of variables. X and Y are *independent given Z in context $C=\gamma$* if

$$P(X | Z, Y, C=\gamma) = P(X | Z, C=\gamma)$$

Figure 1: A Bayesian network.

whenever $P(Y, Z, C=\gamma) > 0$. When Z is empty, one simply says that X and Y are *independent in context* $C=\gamma$.

The following example of CSI first appeared in Shafer (1996). Consider three variables: gender, age, and number of pregnancies. Number of pregnancies depends on age for females. It does not depend on age for males. In other words, number of pregnancies is independent of age in the context “gender=male”.

Here is another example. Consider four variables: income, profession, weather, and qualification. A farmer’s income depends on weather and typically does not depend on his qualification. On the other hand, a office clerk’s income depends on his qualification and typically does not depend on weather. In other words, income is independent of qualification in the context “profession=farmer” and it is independent of weather in the context “profession=office-clerk”.

Boutilier *et al* (1996) point out that one can use CSI to come up with compact representations of conditional probabilities. In our running example, suppose all variables are binary and suppose the conditional probability $P(x_3|x_1, x_2)$ is represented by the tree shown in Figure 2 (1). The tree consists of 8 paths. Because x_3 is independent of x_2 in context $x_1=0$, the conditional probability can be represented more compactly by the tree in Figure 2 (2), which consists of 6 paths.

This paper takes a different view on the structures that CSI has to offer. We regard CSI as providing one with opportunities to decompose conditional probabilities into smaller components. In the above example, the fact that x_3 is independent of x_2 in context $x_1=0$ enables one to decompose the conditional probability $P(x_3|x_1, x_2)$ into the two trees shown in Figure 2 (3) (in a sense to be formally defined in the next section).

4 Partial functions

Before showing why CSI can be helpful in inference, this section formally defines the notion of partial functions and introduces an operation for manipulating partial functions. We also formally define the concept of decomposition and generalize the notion of factorization.

Figure 2: CSI and compact representations of conditional probabilities.

Let X be a set of variables. A *partial function* of X is a mapping from a proper subset of possible values of X to the real line. In other words, it is defined only for some but not all possible values of X . The set of possible values of X for which a partial function is defined is called the *domain* of the partial function. A *full function* of X is a mapping from the set of all possible values of X to the real line. In other words, it is defined for all possible values of X . In the rest of a paper, we use the term “function” when we are not sure whether the function under discussion is a partial function or a full function.

The tree f_1 in Figure 2 (3) represents a partial function of variables x_1 and x_3 . It consists of two *paths* $[x_1=0, x_3=0]$ and $[x_1=0, x_3=1]$, each representing a context in which the partial function is defined. The numbers at the leaf nodes are the values for the contexts. We also refer to them as *values of the paths*. The partial function is not defined in all other contexts, namely $[x_1=1, x_3=0]$ and $[x_1=1, x_3=1]$.

4.1 Union-products

Suppose X , Y , and Z are three disjoint sets of variables and suppose $g(X, Y)$ and $h(Y, Z)$ are two functions. The *union-product* $g \uplus h$ of g and h is the function of variables in $X \cup Y \cup Z$ given by

$$(g \uplus h)(X, Y, Z) = \begin{cases} \text{undefined} & \text{when both } g(X, Y) \text{ and } h(Y, Z) \text{ are undefined} \\ g(X, Y) & \text{when } g(X, Y) \text{ is defined and } h(Y, Z) \text{ is undefined} \\ h(Y, Z) & \text{when } g(Y, Z) \text{ is undefined and } h(X, Y) \text{ is defined} \\ g(X, Y)h(Y, Z) & \text{when both } g(X, Y) \text{ and } h(Y, Z) \text{ are defined} \end{cases} \quad (3)$$

The operation is illustrated in Figure 3. We sometimes write $g \uplus h$ as $g(X, Y) \uplus h(Y, Z)$ to make explicit the arguments of f and g . When the domains of g and h are disjoint, we call

Figure 3: Union-product: The two circles in the left figure represent the domains of two functions g and h . The domain of the union-product $g\mathbin{\text{\textcircled{+}}}h$ is the union of those of g and h . The union-product equals the product of g and h in the areas where both g and h are defined; it equals g in the area where only g is defined; and it equals h in the area where only h is defined.

$g\mathbin{\text{\textcircled{+}}}h$ the *union* of g and h and write it as $g\cup h$.

Here are some of the properties of the union-product operation. First, the union-product of two full functions is simply their product. Together with the definition of union, this explains the term union-product. Second, the union-product of a full function and any other function is a full function. Third, the union-product operation is associative and commutative. We use $\mathbin{\text{\textcircled{+}}}\mathcal{F}$ to denote the union-product of a list $\mathcal{F}$ of functions. When functions in $\mathcal{F}$ have disjoint domains, we denote it by $\cup\mathcal{F}$. Finally, suppose g is a full function that contains variable z as an argument. Then $\sum_z(g\mathbin{\text{\textcircled{+}}}h)=(\sum_z g)\mathbin{\text{\textcircled{+}}}h$ for any other function h that does not contain z .

4.2 Decompositions

A list $\mathcal{F}$ of functions with disjoint domains is a *decomposition* of a function f if $f=\cup\mathcal{F}$. A decomposition is *proper* if no two functions in the decomposition share the same set of arguments. A decomposition of a function f is *nontrivial* if there is at least one that has fewer arguments than f itself. A function is *decomposable* if it has a nontrivial decomposition.

The conditional probability shown in Figure 2 (1) is decomposable. The two partial functions shown in Figure 2 (3) constitute a nontrivial proper decomposition of the conditional probability.

Note that any function constitutes a trivial decomposition of itself.

4.3 Union-product factorizations

A list $\mathcal{F}$ of functions is a *union-product factorization*, or simply a *UP-factorization*, of a function f if $f=\mathbin{\text{\textcircled{+}}}\mathcal{F}$. Note that functions in a decomposition must have disjoint domains whereas domains of functions in a UP-factorization might intersect. A decomposition is a UP-factorization but not vice versa.

For any variable z , let $\mathcal{F}_z$ be the set of functions in $\mathcal{F}$ that contain z as an argument. A UP-factorization $\mathcal{F}$ is *normal* if $\mathbb{U}\mathcal{F}_z$ is a full function whenever $\mathcal{F}_z \neq \emptyset$. The following theorem lays the foundation for making inference with UP-factorizations.

Theorem 1 *Suppose $\mathcal{F}$ is a normal UP-factorization of a full function f and z is an argument of f . Then $\mathcal{G} = (\mathcal{F} \setminus \mathcal{F}_z) \cup \{\sum_z \mathbb{U}\mathcal{F}_z\}$ is a normal UP-factorization of $\sum_z f$.*

Proof: Because $\mathcal{F}$ is a UP-factorization of f and because the union-product operation is commutative and associative, we have

$$\sum_z f = \sum_z \mathbb{U}\mathcal{F} = \sum_z \{[\mathbb{U}\mathcal{F}_z] \mathbb{U}[\mathbb{U}(\mathcal{F} \setminus \mathcal{F}_z)]\}.$$

Since $\mathcal{F}$ is normal, $\mathbb{U}\mathcal{F}_z$ is a full function. Moreover no functions in $\mathcal{F} \setminus \mathcal{F}_z$ contain z , hence $\mathbb{U}(\mathcal{F} \setminus \mathcal{F}_z)$ does not contain z . By the fourth property of the union-product, we have

$$\sum_z f = [\sum_z \mathbb{U}\mathcal{F}_z] \mathbb{U}[\mathbb{U}(\mathcal{F} \setminus \mathcal{F}_z)] = \mathbb{U}\mathcal{G}.$$

In words, $\mathcal{G}$ is a UP-factorization of $\sum_z f$.

To show that $\mathcal{G}$ is normal, let x be a variable such that $\mathcal{G}_x \neq \emptyset$. If $\mathcal{F}_x$ and $\mathcal{F}_z$ are disjoint, then $\mathcal{G}_x = \mathcal{F}_x$ and hence $\mathbb{U}\mathcal{G}_x$ is a full function. If $\mathcal{F}_x$ and $\mathcal{F}_z$ are not disjoint, on the other hand, then $\mathcal{G}_x = (\mathcal{F}_x \setminus \mathcal{F}_z) \cup \{\sum_z \mathbb{U}\mathcal{F}_z\}$. Since $\sum_z \mathbb{U}\mathcal{F}_z$ is a full function, $\mathbb{U}\mathcal{G}_x$ must also be a full function by the second property of the union-product operation. The theorem is therefore proved. $\square$

5 CSI and inference

This section explains why CSI can be helpful in inference and presents a high level inference algorithm that takes advantage of CSI.

5.1 Finer-grain factorization

A BN factorizes a joint probability into a list of conditional probabilities. CSI can be helpful in inference because it allows one to further decompose some of the conditional probabilities into smaller pieces. In our running example, we have assumed earlier that x_3 is independent of x_2 in context $x_1=0$ and that the conditional probability $P(x_3|x_1, x_2)$ is decomposed into the two partial functions f_1 and f_2 shown in Figure 2 (3). Now further assume x_5 is independent of x_3 given x_4 in context $x_1=0$ and x_5 is independent of x_4 given x_3 in context $x_1=1$. Also assume $P(x_5|x_1, x_3, x_4)$ is decomposed into the two partial functions f_3 and f_4 shown in Figure 4 (1) because of CSI. Then we get the following list of functions:

$$P(x_1), P(x_2), f_1, f_2, P(x_4|x_1, x_2), f_3, f_4. \quad (4)$$

Figure 4: Decomposition of $P(x_5|x_1, x_3, x_4)$ and evidence absorption.

Those functions constitute a UP-factorization of the joint probability $P(x_5|x_1, x_3, x_4)$ because

$$\begin{aligned}
& P(x_1) \uplus P(x_2) \uplus f_1 \uplus f_2 \uplus P(x_4|x_1, x_2) \uplus f_3 \uplus f_4 \\
&= P(x_1) \uplus P(x_2) \uplus [f_1 \uplus f_2] \uplus P(x_4|x_1, x_2) \uplus [f_3 \uplus f_4] \\
&= P(x_1) \uplus P(x_2) \uplus P(x_3|x_1, x_2) \uplus P(x_4|x_1, x_2) \uplus P(x_5|x_1, x_3, x_4) \\
&= P(x_1, x_2, x_3, x_4, x_5),
\end{aligned}$$

where the first equation is true because the union-product operation is associative and the third equation is true due to the first property of the operation. The factorization is also normal. For example, $\mathcal{F}_{x_3} = \{f_1, f_2, f_4\}$. Since $f_1 \uplus f_2 = P(x_3|x_1, x_2)$ is a full function, so must be $\mathcal{F}_{x_3}$ by the second property of union-product.

Note that this UP-factorization is of finer-grain than the multiplicative factorization shown by (2). In the multiplicative factorization, one has the full function $P(x_5|x_1, x_3, x_4)$, which requires 16 numbers to specify. In the UP-factorization, the full function is replaced by partial functions f_3 and f_4 . The two partial functions require only 8 numbers to specify.

In general, let $\mathcal{F}$ be the set that consists of, for each variable in a BN, the conditional probability of the variable or, when the conditional probability is decomposed, its components. Then $\mathcal{F}$ is a normal UP-factorization of the joint probability of all variables in the BN. It is of finer-grain than the multiplicative factorization given by the BN if at least one conditional probability is decomposed.

5.2 Evidence absorption

Let $h(z, X)$ be a function of variable z and of variables in a set X . For any value α of z , setting z to α in h results in a new function, denoted by $h|_{z=\alpha}$, of variables in X . For any value β of X , $h|_{z=\alpha}(X=\beta)$ is defined if and only if $h(z=\alpha, X=\beta)$ is defined. When this

is the case, it equals $h(z=\alpha, X=\beta)$. Note that $h_{|z=\alpha}$ is a function of variables in X only; variable z is not an argument of $h_{|z=\alpha}$. If h is full function, so is $h_{|z=\alpha}$. For convenience, let $h_{|z=\alpha}$ be h itself when h does not contain z as an argument.

Suppose a list R of variables are observed and we are interested in the posterior probability $P(Q|R=R_0)$ of a set Q of query variables after obtaining evidence $R=R_0$, where R_0 is the list of observed values. To compute the posterior probability, the first step is to absorb evidence. *Absorbing evidence* means to set observed variables to their observed values in all functions of $\mathcal{F}$. Denote the resulting list of functions by `absorbEvidence`($\mathcal{F}, R, R_0$).

In our running example, suppose x_5 is observed to take value 1. After absorbing this piece of evidence, the partial functions f_3 and f_4 become the partial function f'_3 and f'_4 shown in Figure 4 (2). There are no changes otherwise.

5.3 The variable elimination algorithm

Let Z be the set of variables outside $Q \cup R$ and use $h_{|R=R_0}$ to denote the function obtained by setting the observed variables to the observed values in a function h . Since $\mathcal{F}$ is a normal UP-factorization of the joint probability $P(Z, Q, R)$, `absorbEvidence`($\mathcal{F}, R, R_0$) is a normal UP-factorization of the function $P(Z, Q, R)_{|R=R_0}$. Because of Theorem 1, the posterior probability $P(Q|R=R_0)$ can be obtained using the following procedure.

Procedure `VE-CSI`($\mathcal{F}, Q, R, R_0$):

1. $\mathcal{F} \leftarrow \text{absorbEvidence}(\mathcal{F}, R, R_0)$.
2. **For** each variable z outside $Q \cup R$,
3. $g \leftarrow \boxtimes \mathcal{F}_z$;
4. $h \leftarrow \sum_z g$;
5. $\mathcal{F} \leftarrow (\mathcal{F} \setminus \mathcal{F}_z) \cup \{h\}$.
6. $f \leftarrow \boxtimes \mathcal{F}$.
7. Return $f(Q) / \sum_Q f(Q)$ (Renormalization).

The operations carried out at lines 3-5 are usually referred to as *eliminating variable z* . In particular, line 3 *combines* all functions that contain z using the union-product operator and line 4 *sums out* z from the combination. The algorithm is named `VE-CSI` because it eliminates variables outside $Q \cup R$ one by one and it exploits the finer-grain factorization induced by CSI.

5.4 Discussions

When the first input is a multiplicative factorization instead of a UP-factorization, the union-products at lines 3 and 6 reduce to multiplications and `VE-CSI` therefore reduces to the `VE` algorithm (Zhang and Poole 1994, 1996) or equivalently the bucket elimination algorithm (Dechter 1996). Like `VE`, the complexity of `VE-CSI` is heavily influenced by the ordering by which variables are eliminated. Assume functions are represented as trees.

Figure 5: Elimination of variable x_3 .

Define the *size* of a function to be the number of paths in its tree representation. We suggest the following heuristic: as the next variable to eliminate, choose the variable such that the sum of the sizes of functions that contain the variable is the minimum.

In both VE and VE-CSI, the elimination of a variable involves those and only those functions that contain the variable. The input functions of VE-CSI are of finer-grain than the input functions of VE. Consequently, VE-CSI deals with fewer numbers than VE when eliminating a variable.

As an example, consider the problem of computing the posterior probability $P(x_2|x_5=1)$ in our running example. To solve the problem, VE-CSI starts with the UP-factorization given by (4). After absorbing evidence, f_3 and f_4 becomes f'_3 and f'_4 (Figure 4 (2)). Suppose the first variable to eliminate is x_3 . To eliminate x_3 , VE-CSI first combines the partial functions f_1 , f_2 , and f'_4 . The amount of numbers involved is $2+4=6$. On the other hand, VE starts with the multiplicative factorization given by (2). To eliminate x_3 , it needs to combine $P(x_3|x_1, x_2)$ and $P(x_5|x_1, x_3, x_4)_{|x_5=1}$. This involves $8+8=16$ numbers.

This is a good place to discuss the advantage of our view on the structures CSI has to offer over the view advocated by Boutilier *et al* (1996). Using CSI, we have decomposed the conditional probability $P(x_5|x_1, x_3, x_4)$ into two partial functions f_3 and f_4 , the first of which is not required when eliminating variable x_3 because it does not contain x_3 as an argument. Boutilier *et al*, on the other hand, would represent the conditional probability compactly as *one* tree, which is a simply the merge of f_3 and f_4 (see Figure 2 for another example). While the tree has the same representational complexity as the two partial functions, it makes inference less efficient; The elimination of x_3 requires the entire tree, even though the portion of the tree that corresponds to f_3 is not really needed.

6 Preserving structures during inference

The algorithm **VE-CSI**, as it stands, is rather abstract. We will make it more concrete in this and next two sections. While doing so, a major issue we consider is how to preserve the structures as much as possible during inference. To begin with, this section explains precisely what we mean by preserving structures and discusses the implications.

VE-CSI eliminates a variable in two steps. It first combines all functions that contain the variable and then sums out the variable from the combination. Two full functions are produced, one at each step. The two full functions are often decomposable. To eliminate x_3 in the running example, **VE-CSI** first computes the union-product of f_1 , f_2 , and f'_4 and then calculates $\sum_{x_3} f_1 \uplus f_2 \uplus f'_4$. As it turns out, the three partial functions in Figure 5 (1) constitute a decomposition of the union-product $f_1 \uplus f_2 \uplus f'_4$ and the two partial functions in Figure 5 (3) constitute a decomposition of $\sum_{x_3} f_1 \uplus f_2 \uplus f'_4$.

By preserving structures we mean that, when the aforementioned two full functions are decomposable, we find a decomposition of each of them instead computing the functions themselves. It is desirable to preserve structures for two reasons. First, decomposition simplifies future computations. Second, it is often less expensive to obtain a decomposition of a function than to compute the function itself.

There is one complication associated with structure preserving. As can be seen from the proof of Theorem 1, the correctness of **VE-CSI** relies on the fact that the union-product of all functions that contain a variable is a full function. We have not been able to either prove or disprove whether the fact remains true when structures are preserved. To sidestep the issue, we introduce a special function **1**. It is a full function with no arguments and takes value 1. When represented as tree, **1** consists of only one node that stores its value 1. It is evident that adding copies of function **1** to the list of functions does not affect the output of **VE-CSI**. However, including one copy of **1** to the list of functions that contain a certain variable guarantees their union-product be a full function due to the second property of the union-product operation.

It would be a weakness if the sole purpose of introducing function **1** is to sidestep the unresolved issue mentioned above. Fortunately, this is not case. As will be explained later, the function is also needed for algorithmic simplicity and efficiency.

VE-CSI needs to be modified in order to preserve structures. At line 3, one should find a decomposition of the union-product of function **1** and functions that contain the variable being eliminated, instead of computing the union-product itself. A procedure for this task will be developed in the next section. It is named **decompUP** for obvious reasons. At line 4, one should start with the decomposition obtained at line 3 and find a decomposition of the function h . A procedure for this task will be developed in Section 8. It is named **decompSumOut** again for obvious reasons. Finally, the singleton set $\{h\}$ at line 5 should be replaced by the decomposition found at line 4. After the modifications, **VE-CSI** reads as follows.

Procedure **VE-CSI**($\mathcal{F}, Q, R, R_0$):

1. $\mathcal{F} \leftarrow \text{absorbEvidence}(\mathcal{F}, R, R_0)$.
2. **For** each variable z outside $Q \cup R$,
3. $\mathcal{G} \leftarrow \text{decompUP}(\mathcal{F}_z \cup \{\mathbf{1}\})$;
4. $\mathcal{H} \leftarrow \text{decompSumOut}(\mathcal{G}, z)$;
5. $\mathcal{F} \leftarrow (\mathcal{F} \setminus \mathcal{F}_z) \cup \mathcal{H}$.
6. $f \leftarrow \mathbb{W}\mathcal{F}$.
7. Return $f(Q) / \sum_Q f(Q)$.

7 Finding decompositions of union-products

This section shows how to find a proper decomposition of the union-product of a list of functions. As discussed in the previous section, we can assume one of the functions is $\mathbf{1}$. The problem is solved if we can solve the following special version of the problem:

Given a proper decomposition $\mathcal{G}$ of a full function g , find a proper decomposition of the union-product of g and another function h .

To see why this is the case, let $\text{decompUP0}(\mathcal{G}, h)$ be a routine that solves the special problem. Then a proper decomposition of the union-product of a list $\mathcal{F}$ of functions, one of which being $\mathbf{1}$, can be found using the following procedure.

Procedure **decompUP**($\mathcal{F}$):

1. $\mathcal{G} \leftarrow \{\mathbf{1}\}, \mathcal{F} \leftarrow \mathcal{F} \setminus \{\mathbf{1}\}$.
2. **For** each function $h \in \mathcal{F}$,
3. $\mathcal{G} \leftarrow \text{decompUP0}(\mathcal{G}, h)$.
4. Return $\mathcal{G}$.

The rest of this section is devoted to the special version of the problem. We begin by introducing a new operation for manipulating partial functions.

7.1 Half union-products

Suppose $g(X, Y)$ and $h(Y, Z)$ are two functions. The *half union-product* $g \mathbb{E} h$ of g with h is the function of variables in $X \cup Y \cup Z$ given by

$$(g \mathbb{E} h)(X, Y, Z) = \begin{cases} \text{undefined} & \text{when } g(X, Y) \text{ is undefined} \\ g(X, Y) & \text{when } g(Y, Z) \text{ is defined and } h(X, Y) \text{ is undefined} \\ g(X, Y)h(Y, Z) & \text{when both } g(X, Y) \text{ and } h(Y, Z) \text{ are defined} \end{cases} \quad (5)$$

This operation is illustrated in Figure 6. Here are some of its properties. First, it is not commutative; $g \mathbb{E} h$ is not the same as $h \mathbb{E} g$ except when g and h share the same domain.

Figure 6: Half union-products: The two circles in the left figure represent the domains of two functions g and h . The half union-product $g \mathbin{\&E} h$ is defined only over the domain of g . It equals the product of g and h in the area where both g and h are defined and it equals g in the area where only g is defined.

Second, if the domain of h is a subset of that of g , then $g \mathbin{\&E} h = g \mathbin{\&U} h$. In particular, this is true for any function h when g is a full function. Third, if the domains of g and h are disjoint, then $g \mathbin{\&E} h = g$. The same is true when the values of g are all zero or the values of h are all one. Fourth, $g \mathbin{\&E} (h_1 \mathbin{\&U} h_2) = (g \mathbin{\&E} h_1) \mathbin{\&E} h_2$ for any three functions g , h_1 , and h_2 . Finally, if two functions g_1 and g_2 have disjoint domains, then so do the half union-products $g_1 \mathbin{\&E} h$ and $g_2 \mathbin{\&E} h$ for any other function h and $(g_1 \cup g_2) \mathbin{\&E} h = (g_1 \mathbin{\&E} h) \cup (g_2 \mathbin{\&E} h)$.

7.2 Solving the special version of the problem

Let g and h be two functions. Suppose g is a full function and $\mathcal{G}$ is a proper decomposition of g . To see why the union-product $g \mathbin{\&U} h$ might be decomposable, enumerate functions in $\mathcal{G}$ as $g_1, g_2, \dots, g_n$. By the second and fifth properties of the half union-product operation, we have

$$g \mathbin{\&U} h = g \mathbin{\&E} h = (\cup_{i=1}^n g_i) \mathbin{\&E} h = \cup_{i=1}^n (g_i \mathbin{\&E} h).$$

Since the decomposition $\mathcal{G}$ is proper, the sets of variables in different g_i 's are different. Consequently, the sets of variables in the half union-products $g_i \mathbin{\&E} h$ might be different for different g_i 's. When this is a case, one of the half union-product involves fewer variable than $g \mathbin{\&U} h$ and hence $g \mathbin{\&U} h$ is decomposable.

There is another reason why the union-product might be decomposable; some of the half union-products themselves might be decomposable. We will show this in the next two subsections and develop a routine, named `decompHalfUP`, for finding a proper decomposition of a half union-product. Using this routine, we can find a proper decomposition of $g \mathbin{\&U} h$ as follows.

Procedure **decompUP0**($\mathcal{G}, h$):

1. $\mathcal{H} \leftarrow \emptyset$.
2. **For** each function $g_i \in \mathcal{G}$,
3. $\mathcal{H} \leftarrow \mathcal{H} \cup \text{decompHalfUP}(g_i, h)$.
4. Return **merge**($\mathcal{H}$).

The subroutine **merge**($\mathcal{H}$) merges each group of functions in $\mathcal{H}$ with the same set of variables into one function. This is to make the decomposition $\mathcal{H}$ of $g \bowtie h$ proper. The reasons for keeping decompositions proper will be discussed at the end of this section.

Note that finding a decomposition of $g \bowtie h$ is more difficult if g is a partial function. To appreciate the difficulty, define the *set-difference* $h \ominus g$ of h and g be the function that is defined only in the area where h is defined and g is not and that equals h in the area. Then we have

$$g \bowtie h = [g \boxtimes h] \cup [h \ominus g] = [\cup_{i=1}^n (g_i \boxtimes h)] \cup [h \ominus g].$$

So in addition to the half union-products $g_i \boxtimes h$, we would also need to deal with the set-difference $h \ominus g$. The introduction of function **1** relieves us from the need to consider set-differences.

7.3 Computing half union-products

To see why half union-products might be decomposable and to gain insights about how decompositions of half union-products can be obtained, this subsection discusses how half union-products themselves can be computed.

7.3.1 The brute-force approach

Assume functions are represented as trees. For convenience, we do not clearly distinguish between functions and their tree representations. When we say a path of a function, we mean a path in its tree representation.

For any path p in a tree, use **val**(p) to denote its value and **var**(p) to denote the set of variables it involves. Suppose p consists of assignments $[x_1=\alpha_1, \dots, x_k=\alpha_k]$. Let x be variable outside **var**(p). *Expanding path p on variable x* means to replace p with a number of new paths, one for each possible value β of x , that have the same value as p and that consist of assignments $[x_1=\alpha_1, \dots, x_k=\alpha_k, x=\beta]$. It is evident that path expansion does not change the function that a tree represents.

Suppose g and h are two functions and Z is the set of variables that are arguments of h but not of g . Here is the brute-force approach for obtaining a tree representation of the half union-product $g \boxtimes h$.

1. Expand paths of g on all variables in Z . Afterwards, each path of g involves all variables in Z and hence can be compatible with at most one path of h .

Figure 7: Decompositions of half union-products. See text for explanations.

2. For each path p of g that is compatible with a path q of h , change its value to $\text{val}(p) \times \text{val}(q)$.

Consider the partial functions f_{10} and f_{11} shown in Figure 7, where all variables are binary. Using the brute-force method to find a tree representation of $f_{10} \mathbb{E} f_{11}$, one obtains the tree f_{12} . It is evident that the half union-product can be represented more parsimoniously by tree f_{14} . We next show how to find this more parsimonious representation. The key is to avoid unnecessary path expansions.

7.3.2 Finding parsimonious tree representations of half union-products: A special case

Consider the situation where h consists of only one path q . When the value of q is 1, $g = g \mathbb{E} h$ according to the third property of the half union-product operation. Hence the tree representation of g is a tree representation of $g \mathbb{E} h$. There is no need to expand any paths.

To deal with the case when the value of q is not 1, suppose q consists of assignments $[y_1 = \alpha_1, y_2 = \alpha_2, \dots, y_k = \alpha_k]$. We claim that a tree representation of $g \mathbb{E} h$ can be obtained from the tree representation of g as follows:

For each path p of g that is compatible with q and $\text{val}(p) \neq 0$,

1. Expand it using the following routine:

Procedure **expandPath**(p, q):

1. **For** $i=1$ to k ,
2. **If** $y_i \in \text{var}(q) \setminus \text{var}(p)$,
3. expand p on y_i , and
4. $p \leftarrow$ the new path that contains assignment $y_i = \alpha_i$.
5. Return p .

2. Change the value of the path returned to $\text{val}(p) \times \text{val}(q)$.

This procedure avoids unnecessary path expansions. First, original paths of g that have value zero are not expanded. Second, original paths of g that are not compatible with q are not expanded either. Third, among all the new paths created by **expandPath**, those incompatible with q are not further expanded.

To show the correctness of the procedure, ignore the second step for the time being. Then the resultant tree still represents function g . Divide the tree into two components g_1 and g_2 such that g_2 consists of all the paths that were once returned by **expandPath** and g_1 consists of all other paths. Further divide g_1 into g_{10} and g_{11} such that g_{10} consists of all the zero-valued paths and g_{11} consists of all other paths. Then $g = g_{10} \cup g_{11} \cup g_2$. Because of the fifth and third properties of the half union-product operation and the fact that all paths in g_{10} have value zero, we have

$$g \mathbb{E} h = (g_{10} \cup g_{11} \cup g_2) \mathbb{E} h = g_{10} \cup [(g_{11} \cup g_2) \mathbb{E} h]$$

Next consider a path of g_{11} . If it is one of the original paths of g , then it cannot be compatible with q . Otherwise, it would have been expanded since its value is not 0. If it is one of paths created at line 3 of **expandPath**, it cannot be compatible with q either. Otherwise, it would have been either further expanded or returned. So paths of g_{11} are not compatible with q . Again by the fifth and third properties of the half union-product operation, we have

$$g \mathbb{E} h = g_{10} \cup [(g_{11} \cup g_2) \mathbb{E} h] = (g_{10} \cup g_{11}) \cup (g_2 \mathbb{E} h).$$

The equation implies that a tree representation of $g \mathbb{E} h$ can be obtained by first finding a tree representation of $g_2 \mathbb{E} h$ and then adding paths of g_{10} and g_{11} to the tree. Since each path of g_2 is compatible with q and involves all the variables y_i , a tree representation of $g_2 \mathbb{E} h$ can be obtained by simply multiplying $\text{val}(q)$ to the value of each path of g_2 . But this is exactly what happens at step 2. The correctness of the procedure is therefore proved.

7.3.3 Finding parsimonious tree representations of half union-products: The general case

Now consider the general case when h consists of m paths $q_1, q_2, \dots, q_m$, where $m \geq 1$. Each path can be viewed as a partial function by itself. Because of the fourth property of

the half union-product operation, we have

$$g \mathbb{E} h = g \mathbb{E} (\cup_{i=1}^m q_i) = (\dots ((g \mathbb{E} q_1) \mathbb{E} q_2) \dots) \mathbb{E} q_m.$$

Consequently, a tree representation $g \mathbb{E} h$ can be obtained by first finding a tree representation of $g \mathbb{E} q_1$, then a tree representation of $(g \mathbb{E} q_1) \mathbb{E} q_2$, and so on. This is the idea behind the following algorithm.

Procedure **halfUP**(g, h):

1. **For** each path q of h such that $\text{val}(q) \neq 1$,
2. **For** each path p of g that is compatible with q and $\text{val}(p) \neq 0$,
3. $p' \leftarrow \text{expandPath}(p, q)$.
4. $\text{val}(p') \leftarrow \text{val}(p') \times \text{val}(q)$.
5. Return g .

Let us trace the computations that **halfUP** carries out when called to find a tree representation of $f_{10} \mathbb{E} f_{11}$. At the beginning, q is set to be the first path of f_{11} . The first and the second paths of f_{10} are compatible with q . Lines 3 and 4 are skipped for the latter since its value is 0. The subroutine **expandPath** is called when p is the former. At line 3 of **expandPath**, p is expanded on variable x_4 , resulting two new paths $[x_1=0, x_3=0, x_4=0]$ and $[x_1=0, x_3=0, x_4=1]$. At line 4, p is reset to be the first new path because $x_4=0$ in path q . The path is further expanded on x_5 at line 3, resulting in two new paths $[x_1=0, x_3=0, x_4=0, x_5=0]$ and $[x_1=0, x_3=0, x_4=0, x_5=1]$. At line 4, p is reset to be the first new path because $x_5=0$ in path q . The value of the path is then changed to $0.3 \times 0.1 = 0.03$ at line 4 of **halfUP**. After all those operations, f_{10} becomes f_{13} .

Next, q is set to be the second path of f_{11} . The second and the fourth path of f_{13} are compatible with q . Lines 3 and 4 of **halfUP** are skipped for the latter since its value is zero. The subroutine **expandPath** is called when p is the former. Since $\text{var}(q) \setminus \text{var}(p) = \emptyset$, **expandPath** does nothing. At line 4 of **halfUP**, the value of p is changed to $0.3 \times 0.9 = 0.27$. Afterwards, f_{10} further becomes f_{14} , which represents the half union-product $f_{10} \mathbb{E} f_{11}$.

7.4 Finding decompositions of half union-products

The example of the previous subsection reveals why half union-products might be decomposable and how to find their decompositions. Tree f_{14} represents the half union-product $f_{10} \mathbb{E} f_{11}$. Different paths in the tree involve different variables: the first two paths involve variables x_1, x_3, x_4 , and x_5 ; the third path involves variables x_1, x_3 , and x_4 ; while the last two paths involve variables x_1 and x_3 . Splitting the tree, we get the three trees f_{15} , f_{16} , and f_{17} , which constitute a proper decomposition of the half union-product.

In general, the tree returned by **halfUP** often contains paths that involve different variables. Let **split** be a routine that splits such a tree into a minimum number of subtrees such that all paths in each subtree involve the same variables. Then a proper decomposition of the half union-product $g \mathbb{E} h$ can be obtained as follows:

Procedure **decompHalfUP**(g, h):

1. $f \leftarrow \text{halfUP}(g, h)$.
2. Return **split**(f).

7.5 Remarks

The description of **decompUP** is now complete. Here are four remarks on various aspects of the algorithm. The first two remarks are important for better understanding the algorithm and for later expositions. The last two remarks are not essential and should probably be skipped in first reading.

7.5.1 Properness of decompositions

This first remark has to do with the complexity of the the routine **decompUP0**($\mathcal{G}, h$) and the role of properness of decompositions. For simplicity, consider only the case when h consists of one path q . During the execution of **decompUP0**, the subroutine **decompHalfUP** is called once for each function in the the decomposition $\mathcal{G}$. At line 3 of the subroutine, one needs to identify all paths of the functions that are compatible with q . This is the most time consuming step when the total number of paths of functions in $\mathcal{G}$ is large. It is to keep the complexity of the step minimum that we always ensure decompositions be proper.

To see why properness helps, consider, for the sake of argument, the case when all functions in $\mathcal{G}$ are single-path functions. Here one would have to check all paths (functions) in order to find those that are compatible with q . The task becomes much easier if paths that involve the same set of variables are grouped into one tree. As a matter of fact, to find all paths in a tree that are compatible with q , one can start from the root and descend to leaf nodes along selected paths as follows: When a node labelled with a variable x is reached, one descends along a branch $x=\alpha$ of the node only if $x=\alpha$ is compatible with q . The paths whose leaf nodes are reached are those that are compatible with q . This way only a fraction of the paths in the tree are visited.

7.5.2 Pruning zero-valued paths

The second remark has to do with zero-valued paths. Consider the functions in the working list $\mathcal{G}$ during the execution of **decompUP**. Initially there is only one function, namely **1**. It does not contain any zero-valued paths. Functions in $\mathcal{G}$ are modified by the subroutine **halfUP** at lines 3 and 4 and zero-valued paths are only created at line 4 when **val**(q)=0.

As can be seen from line 2 of **halfUP**, zero-valued paths are never expanded. They are simply carried over to the output of **decompUP**. To save space and to simplify the task of identifying compatible paths, we prune all zero-valued paths once they are generated. To do so, we replace line 4 of **halfUP** by the following two lines:

- 4.1. **If** **val**(q)=0, prune p' from g .

4.2. **Else** $\text{val}(p') \leftarrow \text{val}(p') \times \text{val}(q)$.

Before this modification, **decompUP** returns a list of functions that is a proper decomposition of a full function. After the modification, the same list of functions is returned except that all the zero-valued paths have been pruned. From now on, we assume the modification has been incorporated.

It should be noted that while one can the prune zero-valued paths created during the execution of **decompUP**, one cannot prune zero-valued paths from its input functions. Each path of a input function, zero-valued or not, is needed at line 3 of **halfUP** (as q). However, one can organize them as follows to make **decompUP** run faster: (1) divide each input function into two functions; a *zero-valued function* consisting of all the zero-valued paths and the other consisting of other paths, and (2) process zero-valued functions first in the for-loop of **decompUP**. This reduces the sizes of the trees generated by **decompUP** since all zero-valued paths are generated and pruned at the beginning

7.5.3 Avoid unnecessary introductions of function **1**

When eliminating a variable z , we might need a copy of the special function **1** for two reasons. When the union-product $\mathbb{U}\mathcal{F}_z$ of functions that contain variable z is not a full function, we need to combine it with **1**. In **decompUP**, we need a decomposition of a full function to start with because of the requirement of the subroutine **decompUP0**.

If there is a subset $\mathcal{F}'$ of $\mathcal{F}_z$ that is a decomposition of a full function, then both reasons vanish. The union-product $\mathbb{U}\mathcal{F}_z$ must be a full function due to the second property of the union-product operation and in **decompUP**, we can start by setting $\mathcal{G} \leftarrow \mathcal{F}'$, $\mathcal{F} \leftarrow \mathcal{F} \setminus \mathcal{F}'$. Consequently, there is no need to introduce a copy of **1** when eliminating variable z .

The problem is how to tell whether there is a subset of $\mathcal{F}_z$ that is a decomposition of a full function. This problem can be solved by a small amount of book keeping and by making use of the following two observations. First, functions obtained by decomposing a conditional probability constitute a decomposition of a full. This remains true after evidence absorption. Second, as will be seen in the next section, the list of functions obtained at line 4 of **VE-CSI** is a decomposition of a full function.

7.5.4 Procedural short-cuts

The last remark has to do with the task of finding a decomposition of the half union-product of the special function **1** with another function h . One can of course simply call the routine **decompHalfUP**. There is a more efficient alternative.

Note that the half union-product $\mathbf{1} \mathbb{E} h$ equals h in the domain of h and takes value 1 elsewhere. A tree representation for $\mathbf{1} \mathbb{E} h$ can hence be obtained by *patching 1's to* the tree representation of h as follows: For each node labelled with a variable, say x , if there is no branch for a value α of x , create a branch for α and set the value of the branch to 1. A proper decomposition can then be found by splitting the tree.

Figure 8: A more efficient method for finding a proper decomposition of the half union-product of function $\mathbf{1}$ with the function f_{11} in Figure 7. The function f_{18} is obtained from f_{11} by patching 1's. Splitting f_{18} , we get the decomposition $\{f_{19}, f_{20}, f_{21}\}$.

As an example, consider the half union-product of $\mathbf{1}$ with the function f_{11} shown in Figure 7. Since there are no branches for $x_1=1$ and $x_4=1$, two new branches are created, resulting in the tree f_{18} shown in Figure 8. The tree represents the half union-product of $\mathbf{1}$ with f_{11} . Splitting it, we get trees f_{19} , f_{20} , f_{21} , which constitute a proper decomposition of the half union-product.

This alternative method for finding a proper decomposition of $\mathbf{1} \otimes h$ is obviously more efficient than `decompHalfUP`, especially the size of h is large.

8 Summing out variables from decompositions

VE-CSI eliminates a variable in two steps. It first combines function $\mathbf{1}$ and functions that contain the variable and then sums out the variable from the combination. In the previous section, we have discussed how structures can be preserved at the first step. This section shows how structures can be preserved at the second step. To this end, we need yet another operation for manipulating functions.

8.1 Union-sums

For any two functions $g(X, Y)$ and $h(Y, Z)$, their *union-sum* $g \uplus h$ is defined in the same way as their union-product except with $g(X, Y)h(Y, Z)$ replaced by $g(X, Y) + h(Y, Z)$ in equation (3).

Here are some of the properties of the union-sum operation. First, it is commutative and associative. For any list $\mathcal{H}$ of function, we use $\uplus \mathcal{H}$ to denote the union-sum of all functions in $\mathcal{H}$. Second, the union-sum of two full functions equals their sum. Third, the union-sum of two functions with disjoint domains equals their union. Finally, the union-sum of a full function with any other function is a full function.

8.2 Problem statement

In the previous section, we have developed an algorithm, namely **decompUP**, for finding a proper decomposition of the union-product $g = \mathbb{U}\mathcal{F}_z \cup \{\mathbf{1}\}$ of function **1** and all functions that contain the variable z being eliminated. The problem in this section is how to find a proper decomposition of $\sum_z g$ from the proper decomposition of g obtained by **decompUP**.

One complication is that **decompUP** does not keep zero-valued paths. To deal with this complication, first consider the list functions that would be returned by **decompUP** if all zero-valued paths were kept. Suppose the list consists of n functions $f_1, f_2, \dots, f_n$. Recall that for any function h and any value α of z , $h|_{z=\alpha}$ denotes the function obtained by setting z to value α in h (Section 5.2). Since the f_i 's constitute a decomposition of g , the functions $f_1|_{z=\alpha}, f_2|_{z=\alpha}, \dots, f_n|_{z=\alpha}$ constitute a decomposition of $g|_{z=\alpha}$, i.e. $g|_{z=\alpha} = \cup_{i=1}^n f_i|_{z=\alpha}$. By the third property of the union-sum operation, we have $g|_{z=\alpha} = \mathbb{U}_{i=1}^n f_i|_{z=\alpha}$.

Now suppose the decomposition actually returned by **decompUP** consists of m functions $g_1, g_2, \dots, g_m$. Consider the union-sum $\mathbb{U}_{i=1}^m g_i|_{z=\alpha}$. It is the same as $\mathbb{U}_{i=1}^n f_i|_{z=\alpha}$ except with all the zero-valued paths pruned. Consequently, it is defined only in the support of $\mathbb{U}_{i=1}^n f_i|_{z=\alpha}$ ³ and within the support, it equals $\mathbb{U}_{i=1}^n f_i|_{z=\alpha}$. Let **0** be the special full function that has no arguments and takes value 0. Then

$$\mathbf{0} \mathbb{U} (\mathbb{U}_{i=1}^m g_i|_{z=\alpha}) = \mathbb{U}_{i=1}^n f_i|_{z=\alpha} = g|_{z=\alpha}.$$

Another way to write $\sum_z g$ is $\sum_\alpha g|_{z=\alpha}$. Because of the second and first properties of the union-sum operation and because of the fact that $\mathbf{0} \mathbb{U} \mathbf{0} = \mathbf{0}$, we have

$$\sum_\alpha g|_{z=\alpha} = \mathbb{U}_\alpha g|_{z=\alpha} = \mathbb{U}_\alpha [\mathbf{0} \mathbb{U} (\mathbb{U}_{i=1}^m g_i|_{z=\alpha})] = \mathbf{0} \mathbb{U} [\mathbb{U}_{i=1}^m (\mathbb{U}_\alpha g_i|_{z=\alpha})]. \quad (6)$$

So the problem of finding a proper decomposition of $\sum_z g$ is really the problem of finding a proper decomposition of the union-sum of function **0** and the functions $g_i|_{z=\alpha}$.

Observe that the set of variables in $g_i|_{z=\alpha}$ is the same for all values α of z . For the same reason as we merge functions with the same arguments when finding decompositions of union-products, it is desirable to compute the union-sum $\mathbb{U}_\alpha g_i|_{z=\alpha}$ first. In the next subsection, we will develop a routine from computing this union-sum directly from g_i . The routine is named **simpleSumOut** for obvious reasons.

A procedure named **decompUS** for finding a proper decomposition of the union-sum of a list of function will be developed in Section 8.3. Using **simpleSumOut** and **decompUS**, we can obtain a proper decomposition of $\sum_z g$ as follows.

Procedure **decompSumOut**($\mathcal{G}, z$):

1. $\mathcal{H} \leftarrow \{\mathbf{0}\}$.
2. **For** each function g_i in $\mathcal{G}$,
3. $\mathcal{H} \leftarrow \mathcal{H} \cup \{\text{simpleSumOut}(g_i, z)\}$.
4. **Return** **decompUS**($\mathcal{H}$).

³For any function $h(X)$, the *support* of h is the set of values of X where h is positive.

Here is a remark about the role of the special function **0**. Even though it might appear paradoxical that we prune zero-valued paths in `decompUP` and introduce function **0** to deal with the consequences, the benefits of doing so is not difficult to appreciate. While many zero-valued paths might be pruned in `decompUP`, function **0**, when represented as a tree, consists of only one zero-valued path. Moreover, function **0** simplifies `decompUS` in the same way as function **1** simplifies `decompUP`.

8.3 Summing out variables from functions

Suppose a path p in a tree consists of assignments $[x_1=\alpha_1, \dots, x_k=\alpha_k]$. Let `delVar`(p, z) be a routine that deletes from p the assignment $x_i=\alpha_i$ when $z=x_i$ and does nothing when $z \neq x_i$ for all i . Use `nil` to denote the partial function that is not defined anywhere. It corresponds to the empty tree. The following procedure computes $\uplus_{\alpha} g_i|_{z=\alpha}$ directly from function g_i .

Procedure `simpleSumOut`(g_i, z):

1. $h \leftarrow \text{nil}$.
2. **For** each path p of g_i ,
3. $q \leftarrow \text{delVar}(p, z)$.
4. **If** q is compatible with a path q' of h ,
5. $\text{val}(q') \leftarrow \text{val}(q') + \text{val}(q)$.
6. **Else** insert q to h .
7. **Return** h .

As an example, consider summing out variable x_3 from the function f_5 shown in Figure 5. At beginning, p is set to the first path of f_5 . Deleting x_3 from p results in the path in f'_5 , which is simply inserted to h since it is `nil`. Next, p is set to the second path of f_5 . Deleting x_3 from p results in the path in f''_5 . It is compatible with the path already inserted to h . Hence line 5 is executed, resulting in function f_8 .

In the special case when g_i does not involve z , $\uplus_{\alpha} g_i|_{z=\alpha}$ can be computed by simply multiplying the number of possible values of z to the value of each path in g_i . This is obviously more efficient than `simpleSumOut`.

8.4 Finding decompositions of union-sums

This subsection shows how to find a proper decomposition of the union-sum of a list of functions. As discussed earlier, we can assume that one of the functions is **0**.

Define the natural exponential e^g and the natural logarithm $\ln(g)$ of a function g in the obvious way⁴. In terms of tree representation, e^g and $\ln(g)$ can be obtained from g by changing the value $\text{val}(p)$ of each path of g to $e^{\text{val}(p)}$ or $\ln(\text{val}(p))$ respectively. It is

⁴The logarithm $\ln(g)$ is well-defined only when g takes only positive values.

easy to see that for any two functions g and h , $g \uplus h = \ln(e^g \uplus e^h)$. Moreover if g and h have disjoint domains, so do $\ln(g)$ and $\ln(h)$ and $\ln(g \cup h) = \ln(g) \cup \ln(h)$.

Suppose $\mathcal{H}$ is a list of functions and one of the functions is $\mathbf{0}$. Let $\mathcal{G} = \{e^g \mid g \in \mathcal{H}\}$. $\mathcal{G}$ contains the function $\mathbf{1}$ because $e^{\mathbf{0}} = \mathbf{1}$. A proper decomposition of the union-product $\uplus \mathcal{G}$ can hence be obtained by using `decompUP`. Suppose the decomposition consists of n functions $h_1, h_2, \dots, h_n$. Then $\{\ln(h_1), \ln(h_2), \dots, \ln(h_n)\}$ is a proper decomposition of the union-sum $\uplus \mathcal{H}$ because

$$\uplus \mathcal{H} = \ln(\uplus \mathcal{G}) = \ln(\cup_{i=1}^n h_i) = \cup_{i=1}^n \ln(h_i).$$

This gives us a procedure for finding a proper decomposition of the union-sum $\uplus \mathcal{H}$. It is obtained by wrapping exponential-logarithm transformations around `decompUP` in that each input function is replaced by its natural exponential before passing to `decompUP` and each output function of `decompUP` is replaced by its natural logarithm.

With an eye on a transformation-free algorithm, observe that the exponential-logarithm transformations do not alter the structures of tree representations of functions in that any function, its natural exponential, and its natural logarithm can be represented by three trees with identical structures. The transformations do affect values of paths in trees. Since the only place where `decompUP` accesses values of paths is in the subroutine `halfUP`, one can wrap exponential-logarithm transformations around each call to this subroutine instead of around `decompUP` itself. To do so, one can simply modify the routine `decompHalfUP` by adding “ $g \leftarrow e^g, h \leftarrow e^h$ ” before line 1 and inserting “ $f \leftarrow \ln(f)$ ” between lines 1 and 3.

It is not difficult to see that modified `decompHalfUP` is equivalent to the routine below.

Procedure `decompHalfUS`(g, h):

1. **For** each path q of h such that $\text{val}(q) \neq 0$,
2. **For** each path p of g that is compatible with q ,
3. $p' \leftarrow \text{expandPath}(p, q)$.
4. $\text{val}(p') \leftarrow \text{val}(p') + \text{val}(q)$.
5. Return `split`(g).

If we define *half union-sum* in a way similar to half union-product, then this routine returns a proper decomposition of the half union-sum of g with h . This explains the name `decompHalfUS`.

Since `decompHalfUS` does not perform any exponential-logarithm transformations, we have obtained a transformation-free algorithm for finding a proper decomposition of $\uplus \mathcal{H}$. An explicit description of the algorithm is given below, where the subroutine `decompUS0` is the same as `decompUP0` except with `decompHalfUP` replaced by `decompHalfUS`.

- Procedure `decompUS`($\mathcal{H}$):
1. $\mathcal{G} \leftarrow \{\mathbf{0}\}, \mathcal{H} \leftarrow \mathcal{H} \setminus \{\mathbf{0}\}$.
 2. **For** each function $h \in \mathcal{H}$,
 3. $\mathcal{G} \leftarrow \text{decompUS0}(\mathcal{G}, h)$.
 4. Return $\mathcal{G}$.

9 Empirical results

Experiments have been conducted to demonstrate the computational benefits of **VE-CSI**. The experiments were designed to compare **VE-CSI** with **VE** — an algorithm that is identical to **VE-CSI** except that it does not exploit CSI (see Section 5.4). This section describes the experiments and reports the results.

9.1 Implementation issues

VE-CSI works with both partial and full functions while **VE** works with only full functions. Partial functions should obviously be presented as trees, while full functions can be represented as either trees or tables. To be consistent, both partial and full functions are represented as trees in the implementation of **VE-CSI**. The choice between tree and table representations is not clear for **VE**. On one hand, tree representation allows **VE** to prune zero-valued paths. On the other hand, table representation provides faster access to function values and requires fewer memory allocations. We tried both representations. In the following, we will use **VE-TABLE** to refer to the implementation of **VE** where full functions are represented as tables and use **VE-TREE** to refer to the implementation where full functions are represented as trees.

9.2 Theoretical comparisons

Before diving into the empirical comparisons, it is worthwhile to discuss the pros and cons of the algorithms theoretically. The main advantage of **VE-CSI** is that its input functions are of finer-grain than those of **VE** and it preserves structures. As a consequence, it deals with less numbers than **VE** when eliminating a variable (see the discussions in Section 5.4). On the other hand, **VE** performs only two simple operations on functions, namely multiplication and summing out a variable from a full function. When represented as trees, the product of two full functions can be computed by (1) finding pairs of compatible paths and (2) for each pair of compatible paths, creating a new path by merging their assignments and setting the value of the new path to be the product of their values. This is simpler than to compute the half union-product of one function with another. To sum out a variable from a full function, the subroutine `simpleSumOut` suffices. In contrast, **VE-CSI** needs to do more work in order to sum out a variable from a decomposition. When full functions are presented as tables, the two operations are even simpler.

VE has another advantage when full functions are represented as trees; it allows the pruning of zero-valued paths from all input functions. In contrast, **VE-CSI** has to keep zero-valued paths of the input functions. Zero-valued paths can be pruned only in the subroutine `decompUP`.

In summary, **VE-CSI** deals with less numbers than **VE** when eliminating a variable. However, the operations it carries out are more complex than those carried out by **VE**.

Figure 9: Representation complexities of conditional probabilities in VE-CSI, VE-TREE, and VE-TABLE.

9.3 The testbed

A BN named Water (Jensen *et al* 1989) was used in the experiments⁵. Water is a model for the biological processes of a water purification plant. It consists of 32 variables. Strictly speaking, conditional probabilities of the variables are not decomposable. To make them decomposable, some of the probability values were modified. The induced errors are upper bounded by 0.05. Using a decision tree-like algorithm (Quinlan 1986), we were able to decompose some of the modified conditional probabilities and thereby reduce their representation complexities drastically⁶.

In all experiments, the inputs to VE-CSI were the decompositions of the modified conditional probabilities while the inputs to VE were the modified conditional probabilities themselves. In VE-TABLE, the conditional probabilities were represented as tables. In VE-TREE, they were represented as trees and the zero-valued paths were pruned. The representation complexities of the conditional probabilities in the three algorithms are shown in Figure 9.

9.4 Experiments and results

The experiments were performed on a SUN ULTRA 1 machine. The task was to eliminate all variables according to a predetermined elimination ordering. In the first experiment, an ordering by Kjærulf was used⁷. The performances of VE-CSI, VE-TREE, and VE-TABLE are shown in Figure 10. The chart on the left shows the amounts of times in CPU seconds that VE-CSI, VE-TREE, and VE-TABLE took to eliminate the first n variables for n running from 0 to 31. We see that VE-CSI significantly outperformed VE-TREE, which in turn significantly outperformed VE-TABLE. In particular, VE-CSI completed the entire elimination process in about 7 seconds, while VE-TREE took about 70 seconds and VE-TABLE took about 460 seconds.

⁵Obtained from a Bayesian network repository at Berkeley.

⁶Modifications of probability values actually take place during the decomposition process. We choose to describe them as two separate steps here for presentation clarity.

⁷The ordering was also obtained from the Berkeley repository

Figure 10: Performances of VE-CSI, VE-TREE, and VE-TABLE on one elimination ordering.

Figure 11: Performances of VE-CSI and VE-TREE across 15 elimination orderings.

To offer an explanation of the differences in performance, the chart on the right compares the sizes of the trees/tables created by VE-CSI, VE-TREE, and VE-TABLE when eliminating each variable. For each variable, the size for VE-CSI is the total number of paths in the trees it created at line 3 when eliminating the variable. The size for VE-TREE is the number of paths in the corresponding tree it created and the size for VE-TABLE is the number the number of cells in the corresponding table it created. Those sizes correspond to clique sizes in the clique tree propagation algorithm (Lauritzen and Spiegelhalter 1988, Jensen *et al* 1990, Shafer and Shenoy 1990). We see that on average the size for VE-CSI was much smaller than that for VE-TREE, which in turn was much smaller than that for VE-TABLE. All the sizes reached maximum when eliminating the 20th variable. At that point, the size for VE-CSI was 10,846, that for VE-TREE was 165,376, and that for VE-TABLE was 5,308,416. This not only explain the better time performance of VE-CSI but also show that VE-CSI requires much less memory than both VE-TREE and VE-TABLE.

Note that VE-TABLE performed slightly better than VE-TREE when eliminating the first ten variables. This is because, as pointed out earlier, it provides faster access to function values and requires fewer memory allocations. It fell behind starting from the 15th variable because the tables it produced were much larger than the trees produced by VE-TREE.

In the second experiment, only **VE-C**SI and **VE-T**REE were considered. We generated 14 new elimination orderings by randomly permuting pairs of variables in Kjærullf’s ordering. A trial was conducted with each ordering. The performances of **VE-C**SI and **VE-T**REE accross all the trials are summarized in Figure 11. We see that **VE-C**SI significantly outperformed **VE-T**ABLE in all trials. The speedup was more than one magnitude on average. Moreover, there are four trials where **VE** was not able to complete due to large memory requirements. On the other hand, **VE-C**SI completed each of those four trials in less than 30 seconds.

10 Related work

CSI has its origins in both the BN literature and the influence diagram literature. In the influence diagram literature, CSI arises in the study of asymmetric decision programs and can be traced back to Olmsted (1983). Olmsted introduces a notion of coalescence within influence diagrams, which is slightly more general than CSI. Fung and Shachter (1990) propose to explicitly represent CSI by associating each variable with a collection of contingencies and decompose the conditional probability of the variable according to the contingencies. Smith *et al* (1993) extend this work so that coalescence and impossible contexts, i.e. contexts with zero probability, can both be represented ⁸. Both of these two papers emphasize more on representational issues and less on inferential issues. They provide no general methods for exploiting and preserving structures induced by CSI.

Several methods for exploiting CSI in BN inference have been proposed. Santos and Shimony (1993, 1996) show that CSI can speed up search-based methods for approximating posterior probabilities. Geiger and Heckerman (1996) propose to make use of CSI at the time when a BN is being constructed. A set of local networks is obtained instead of one global network. The set of local networks is called a Bayesian multinet. One can either carry out inference directly in the multinet or convert the multinet into a global network before making inference. This approach requires a predetermined set of conditioning variables and CSI due to other variables is not exploited. Boutilier *et al* (1996) present two methods. One method makes use of CSI to reduce the size of the cutset in the cutset conditioning algorithm. The cutset conditioning algorithm reduces a loopy BN into polytrees by conditioning on a set of variables. It is usually less efficient than algorithms such as **VE** and clique tree propagation that work on the loopy BN itself. The other method proposed by Boutilier *et al* converts CSI statements into conditional independence statements by introducing auxiliary variables. This method creates many auxiliary variables and structural information can be lost during the conversion (Poole 1997).

The rule-based method proposed by Poole (1997) is closely related to this paper and hence deserves special attention. Whereas our method decomposes a conditional probability into partial functions, Poole’s method decomposes it into rules. In the running example,

⁸Impossible contexts are another main source of asymmetries for decision problems.

we have decomposed $P(x_3|x_1, x_2)$ into the partial functions f_1 and f_2 shown in Figure 2. Poole would decompose it into 6 rules:

$$\begin{array}{ll} x_3=0 \leftarrow x_1=0 : 0.3, & x_3=1 \leftarrow x_1=0 : 0.7, \\ x_3=0 \leftarrow x_1=1, x_2 = 0 : 0.6, & x_3=1 \leftarrow x_1=1, x_2 = 0 : 0.4, \\ x_3=0 \leftarrow x_1=1, x_2 = a : 0, & x_3=1 \leftarrow x_1=1, x_2 = 1 : 1. \end{array}$$

As **VE-CSI**, Poole’s algorithm computes posterior probabilities by eliminating variables one by one.

We argue that our method is conceptually clearer and computationally more efficient than Poole’s method. In our method, the problem of exploiting the finger-grain factorization induced by CSI is divided into two subproblems: (1) how to eliminate a variable in such a way that only those functions that contain the variable are required in the process and (2) how to preserve structures during the elimination process. In Section 5, the first subproblem is solved at a level of abstraction that reveals the essence of the problem. In Section 7, structure preservation opportunities during the computation of union-products are clearly identified and are incorporated to **VE-CSI**. In Section 8, a solution for the problem of preserving structures when summing out variables is derived by making use of its relationship to the same problem for union-products. In Poole’s method, on the other hand, the issues are not clearly separated. There is no formal proof of correctness and the issue of structure preservation is not explicitly dealt with.

Our method is more efficient than Poole method for three reasons. First, the most time consuming operation in inference is to identify compatible rules (paths). As pointed out in Section 7.5.1, this can be done much more efficiently with trees (of paths) than with individual rules. Second, our method preserves structures better than Poole’s method. As pointed out earlier, there are no explicit structure preservation mechanisms in Poole’s method, while our method utilizes every opportunity for structure preservation (see Sections 7 and 8). Finally, our method prunes zero-valued paths when computing union-products, while Poole’s method never prunes zero-valued rules.

11 Conclusions

We have studied the role of context-specific independence in Bayesian network inference. Making use of conditional independence, a Bayesian network factorizes a joint probability into a list of conditional probabilities that involve fewer variables. This paper points out that context-specific independence allows one to further decompose some of the conditional probabilities into partial functions that take fewer numbers to specify, resulting in a finer-grain factorization of the joint probability. An inference algorithm that exploits the finer-grain factorization has been developed. The algorithm preserves structures during inference. Experiments have demonstrated that exploiting context-specific independence with our algorithm can bring about significant computational gains.

Both context-specific independence and independence of causal influence allows one to divide conditional probabilities into smaller pieces. It is conceivable that both types of independence might be present in some applications. It is an open problem how to combine techniques for exploiting these two types of independence.

Acknowledgements

The author would like to thank David Poole for useful discussions, Weihong Zhang for commenting on earlier versions of the paper, and Stephen Lee for helping with the implementations.

References

- [1] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller (1996), Context-specific independence in Bayesian networks, *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pp. 115-123.
- [2] R. Dechter (1996), Bucket elimination: A unifying framework for probabilistic inference, *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pp. 211-219.
- [3] R. M. Fung and R. D. Shachter (1990), Contingent Influence Diagrams, Advanced Decision Systems, 1500 Plymouth St., Mountain View, CA 94043, USA.
- [4] D. Geiger and D. Heckerman (1996), Knowledge representation and inference in similarity networks and Bayesian multimeters, *Artificial Intelligence*, 92, pp. 45-74.
- [5] D. Heckerman (1993), Causal independence for knowledge acquisition and inference, *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pp. 122-127.
- [6] D. Heckerman and J. Breese (1994), A new look at causal independence, *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pp. 286-292.
- [7] R. A. Howard, and J. E. Matheson (1984), Influence Diagrams, *The principles and Applications of Decision Analysis*, Vol. II, R. A. Howard and J. E. Matheson (eds.). Strategic Decisions Group, Menlo Park, California, USA.
- [8] F. V. Jensen, U. Kjærulff, K. G. Olesen, and J. Pedersen (1989), Et forprojekt til et ekspertsystem for drift af spildevandsrensning (An expert system for control of waste water treatment — A pilot project), Technical Report, Judex Datasystemer A/S, Aalborg, Denmark (in Danish).

- [9] F. V. Jensen, K. G. Olesen, and K. Anderson (1990), An algebra of Bayesian belief universes for knowledge-based systems, *Networks*, 20, pp. 637 - 659.
- [10] S. L. Lauritzen and D. J. Spiegelhalter (1988), Local computations with probabilities on graphical structures and their applications to expert systems, *Journal of Royal Statistical Society B*, 50: 2, pp. 157 - 224.
- [11] K. G. Olesen and S. Andreassen (1993), Specification of models in large expert systems based on causal probabilistic networks, *Artificial Intelligence in Medicine* 5, pp. 269-281.
- [12] S. M. Olmsted (1983), Representing and solving decision problems, Ph.D. Dissertation, Department of Engineering-Economic Systems, Stanford University.
- [13] J. Pearl (1988), *Probabilistic Reasoning in Intelligence Systems: Networks of Plausible Inference*, Morgan Kaufmann Publishers, Los Altos, CA.
- [14] D. Poole (1997), Probabilistic partial evaluation: exploiting rule structure in probabilistic inference, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1284-1291.
- [15] J. R. Quinlan (1986), Induction of decision trees, *Machine Learning*, 1, pp. 81-106.
- [16] E. Santos Jr. and S. E. Shimony (1993), Belief updating by enumerating high-probability independence-based assignments, *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pp. 506-513.
- [17] E. Santos Jr. and S. E. Shimony (1996), Exploiting case-based independence for approximating marginal probabilities, *International Journal of Approximate Reasoning*, 14, pp. 25-54.
- [18] G. Shafer (1996), *Probabilistic Expert Systems*, SIAM.
- [19] G. Shafer and P. Shenoy (1990), Probability propagation, *Annals of Mathematics and Artificial Intelligence*, 2, pp. 327-352.
- [20] J. E. Smith, S. Holtzman, and J. E. Matheson (1993), Structuring conditional relationships in influence diagrams, *Operations Research*, 41, No. 2, pp. 280-297.
- [21] N. L. Zhang and D. Poole (1994), A simple approach to Bayesian network computations, *Proceedings of the Tenth Canadian Conference on Artificial Intelligence*, pp. 171-178.
- [22] N. L. Zhang and D. Poole (1996), Exploiting causal independence in Bayesian network inference, *Journal of Artificial Intelligence Research*, 5, pp. 301-328.