

Implementing Belief Function Computations

Rolf Haenni,^{1,*} Norbert Lehmann^{2,†}

¹Center for Junior Research Fellows, University of Konstanz,
D-78457 Konstanz, Germany

²Department of Informatics, University of Fribourg,
CH-1700 Fribourg, Switzerland

This article discusses several implementation aspects for Dempster-Shafer belief functions. The main objective is to propose an appropriate representation of mass functions and efficient data structures and algorithms for the two basic operations of combination and marginalization. © 2003 Wiley Periodicals, Inc.

1. INTRODUCTION

Today's research and applications in the field of quantitative reasoning and decision under uncertainty is dominated by Bayesian networks²¹ and their variants. This is somehow surprising because many situations involving uncertainty can not be represented properly within the classical probability framework. For example, there is no adequate way of representing total ignorance. Another problem is the restriction of Bayesian networks to directed acyclic graphs.

To avoid these difficulties, many alternative approaches have been proposed. One of the most promising alternatives is the theory of *belief functions*, also known as the *Dempster-Shafer theory* or *theory of evidence*. The original work of Dempster⁶ and Shafer²² has been followed by a number of theoretical and practical contributions. For example, *theory of hints*,¹⁴ provides a clear and coherent interpretation of belief functions. Another milestone is the axiomatic justification given by Smets' *transferable belief model* (TBM).³⁰ Finally, *probabilistic argumentation systems* (PAS)^{8,17} introduce a more practical perspective and show how belief functions are obtained from a simple way of combining classical propositional logic (or corresponding extensions) with probability theory.

*Author to whom correspondence should be addressed; e-mail: rolf.haenni@gmx.net.

†e-mail: norbert.lehmann@unifr.ch.

Despite its success as a well-founded and general model of human reasoning under uncertainty, belief functions are rarely used in concrete applications. One of the most significant arguments raised against using belief functions in practice is their relatively high computational complexity, especially in comparison with methods based on classical probability theory. In fact, combining belief functions using Dempster's rule of combination is known to be #P-complete in the number of evidential sources.²⁰ Furthermore, from a more practical perspective, the complexity of computing the marginal of multivariate belief functions depends exponentially on the size of the largest node in the underlying hypertree.^{28,17} Thus, facing serious difficulties of complexity, the main challenge of making the Dempster-Shafer theory more applicable in practice is to develop appropriate computational methods.

There are two ways of making the Dempster-Shafer theory more efficient. First, to overcome the computational limitations, there are several approximation methods, most of which produce lower bounds instead of exact results.^{2,10,19,31,32} More sophisticated methods produce lower and upper bounds, thus allowing for judgment of the quality of the approximation.^{7,9} Second, by carefully improving and optimizing the implementation of belief function computations, efficiency is increased further and performance is improved considerably. By looking at today's literature, it seems that nobody has seriously studied this important issue so far. The most advanced discussion is found in Ref. 17.

The aim of this article was to study several aspects of implementing Dempster-Shafer belief functions. It proposes a sophisticated way of encoding mass functions and describes efficient methods for the basic operations of combination and marginalization. In combination with the aforementioned approximation techniques, this leads to tremendous improvements of performance, which highly increases the competitiveness of the Dempster-Shafer theory in comparison with other quantitative approaches to uncertainty management.

2. MULTIVARIATE DEMPSTER-SHAFER THEORY

The primitive elements of the Dempster-Shafer theory are belief functions bel_φ relative to some given evidence φ .^{1,6,17,18,22,29,30} Other representations of φ are its mass function m_φ or its plausibility function pl_φ . Here, we use the notations $[\varphi]_m$, $[\varphi]_b$, and $[\varphi]_p$ instead of m_φ , bel_φ , and pl_φ , respectively. In accordance with Shafer,²³ we speak of *belief potentials* φ (or *potentials* for short) when no particular representation is specified.

A *multivariate* belief potential φ is defined on a finite set of variables $D = \{x_1, \dots, x_n\}$ called *domain* of φ . We use Φ_D to denote the set of all belief potentials relative to D . Every variable $x_i \in D$ has a corresponding set Θ_{x_i} of possible values. The Cartesian product $\Theta_D = \Theta_{x_1} \times \dots \times \Theta_{x_n}$, which is the set of possible configurations of D , is called *frame of discernment* of φ . If D is not explicitly specified, we use $d(\varphi)$ to denote the domain of φ . The *mass function* $[\varphi]_m : 2^{\Theta_D} \rightarrow [0, 1]$ assigns to every set $X \subseteq \Theta_D$ a value in $[0, 1]$ such that

$$\sum_{X \subseteq \Theta_D} [\varphi(X)]_m = 1. \quad (1)$$

Mass functions are also called *basic probability assignments* (BPAs). Often, another condition $[\varphi(\emptyset)]_m = 0$ is imposed. A belief potential φ for which this additional condition holds is called *normalized*. Otherwise, φ is called *unnormalized* and $c_\varphi = [\varphi(\emptyset)]_m$ is the corresponding *conflicting mass*.

The sets $X \subseteq \Theta_D$ for which $[\varphi(X)]_m \neq 0$ are called *focal sets* or *focal elements*. $\text{FS}(\varphi)$ denotes the set of all focal sets of φ . A belief potential φ usually is represented by the set $\{(F_1, m_1), \dots, (F_k, m_k)\}$ of all pairs (F_i, m_i) with $F_i \in \text{FS}(\varphi)$ and $m_i = [\varphi(F_i)]_m$ (for more details see Section 3).

Belief functions $[\varphi]_b : 2^{\Theta_D} \rightarrow [0, 1]$ and *plausibility functions* $[\varphi]_p : 2^{\Theta_D} \rightarrow [0, 1]$ usually are defined in terms of corresponding mass functions by

$$[\varphi(H)]_b \stackrel{\text{def}}{=} \sum_{X \in H} [\varphi(X)]_m = \sum_{\substack{X \subseteq H \\ X \in \text{FS}(\varphi)}} [\varphi(X)]_m \quad (2)$$

$$[\varphi(H)]_p \stackrel{\text{def}}{=} \sum_{X \cap H \neq \emptyset} [\varphi(X)]_m = \sum_{\substack{X \cap H \neq \emptyset \\ X \in \text{FS}(\varphi)}} [\varphi(X)]_m \quad (3)$$

respectively, for all $H \subseteq \Theta_D$. Note that $[\varphi(\Theta_D)]_b = 1$ and $[\varphi(\emptyset)]_p = 0$. By distributing the corresponding proportions of the conflicting mass c_φ among the nonempty focal sets $\text{FS}(\varphi) \setminus \{\emptyset\}$, *normalized* mass, belief, and plausibility functions can be defined by

$$[\varphi(X)]_M \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } X = \emptyset, \\ \frac{[\varphi(X)]_m}{1 - c_\varphi}, & \text{otherwise} \end{cases} \quad (4)$$

$$[\varphi(H)]_B \stackrel{\text{def}}{=} \sum_{X \subseteq H} [\varphi(X)]_M = \sum_{\substack{X \subseteq H \\ X \in \text{FS}(\varphi)}} [\varphi(X)]_M = \frac{[\varphi(H)]_b - c_\varphi}{1 - c_\varphi} \quad (5)$$

$$[\varphi(H)]_P \stackrel{\text{def}}{=} \sum_{X \cap H \neq \emptyset} [\varphi(X)]_M = \sum_{\substack{X \cap H \neq \emptyset \\ X \in \text{FS}(\varphi)}} [\varphi(X)]_M = \frac{[\varphi(H)]_p}{1 - c_\varphi} \quad (6)$$

respectively. Note that $[\varphi(\emptyset)]_B = [\varphi(\emptyset)]_P = 0$, $[\varphi(\Theta_D)]_B = [\varphi(\Theta_D)]_P = 1$, and $[\varphi(H)]_B \leq [\varphi(H)]_P$ for all $H \subseteq \Theta_D$. Normalization can also be defined as a mapping $\nu: \Phi_D \rightarrow \Phi_D$ from an unnormalized belief potential $\varphi \in \Phi_D$ to a normalized potential $\nu(\varphi) \in \Phi_D$ by

$$[\nu(\varphi)]_m \stackrel{\text{def}}{=} [\varphi]_M \quad (7)$$

The basic operations for belief potentials are *combination* and *marginalization*. Combination corresponds to aggregation. It takes two potentials $\varphi_1 \in \Phi_{D_1}$ and $\varphi_2 \in \Phi_{D_2}$ and produces a new potential $\varphi_1 \otimes \varphi_2 \in \Phi_D$ on domain $D = D_1 \cup D_2$. Usually, combination is defined on mass functions by

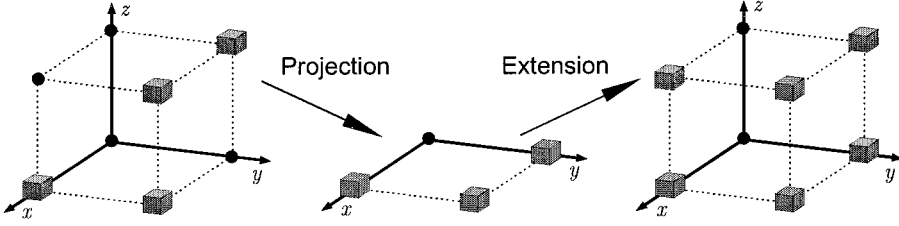


Figure 1. Projection and extension of a set of configurations.

$$\begin{aligned}
 [\varphi_1 \otimes \varphi_2(X)]_m &\stackrel{\text{def}}{=} \sum_{X_1^{\uparrow D} \cap X_2^{\uparrow D} = X} [\varphi_1(X_1)]_m [\varphi_2(X_2)]_m \\
 &= \sum_{\substack{X_1^{\uparrow D} \cap X_2^{\uparrow D} = X \\ X_1 \in \text{FS}(\varphi_1), X_2 \in \text{FS}(\varphi_2)}} [\varphi_1(X_1)]_m [\varphi_2(X_2)]_m \quad (8)
 \end{aligned}$$

where $X_1^{\uparrow D}$ and $X_2^{\uparrow D}$ represent the cylindrical *extensions* of the sets $X_1 \subseteq \Theta_{D_1}$ and $X_2 \subseteq \Theta_{D_2}$ to the new domain D (see Example 1). This way of combining two belief potentials is known as *Dempster's rule of combination*.^{22,a} It relies on the assumption that φ_1 and φ_2 represent *independent* pieces of evidence.

Marginalization takes a belief potential φ on domain D and produces a new potential $\varphi^{\downarrow D'}$ on $C \subseteq D$. It is used to focus the information contained in φ to a smaller domain. It is defined in terms of mass functions by

$$[\varphi^{\downarrow C}(X)]_m \stackrel{\text{def}}{=} \sum_{Y^{\downarrow C} = X} [\varphi(Y)]_m = \sum_{\substack{Y^{\downarrow C} = X \\ Y \in \text{FS}(\varphi)}} [\varphi(Y)]_m \quad (9)$$

where $Y^{\downarrow C}$ denotes the *projection* of the set $Y \subseteq \Theta_D$ to the new domain C .

Example 1. Let $D = \{x, y, z\}$ with $\Theta_x = \Theta_y = \Theta_z = \{0, 1\}$ be the given set of (binary) variables. $\Theta_{\{x,y,z\}} = \{(000), (001), (010), (011), (100), (101), (110), (111)\}$ denotes the set of all configurations. The left side of Figure 1 represents $X = \{(011), (100), (110), (111)\}$ as a cube of which the axes are the variables x , y , and z . $X^{\downarrow \{x,y\}} = \{(01), (10), (11)\}$ is the projection of X to $C = \{x, y\}$. Finally, $Y^{\uparrow D} = \{(010), (011), (100), (101), (110), (111)\}$ is the cylindrical extension of $Y = X^{\downarrow \{x,y\}}$ to the original domain $D = \{x, y, z\}$.

Combination and marginalization satisfy the basic axioms of Shenoy's general framework of *valuation-based systems*.^{24,25,28} When a set $\Psi = \{\varphi_1, \dots, \varphi_r\}$ of several valuations (e.g. belief potentials) on different domains are given, these axioms allow us to write $\varphi_1 \otimes \dots \otimes \varphi_r = \otimes \Psi$ for the combination of all valuations of Ψ and to solve the *problem of inference* $(\otimes \Psi)^{\downarrow C}$ by *local computations*. Originally, this technique was discovered for the case of probabilistic inference.¹⁶

^aNote that Dempster's rule of combination sometimes includes normalization.

A general solution provides Shenoy's *fusion algorithm* and the corresponding propagation techniques for *binary join trees*.^{26,b} Fusion means marginalizing the combination of several valuations to a smaller domain. Without loss of generality, we can consider *binary fusion* $\text{Fus}_C(\varphi_1, \varphi_2) \stackrel{\text{def}}{=} (\varphi_1 \otimes \varphi_2)^{\downarrow C}$ with $C \subseteq D$ and $D = d(\varphi_1) \cup d(\varphi_2)$ as the basic operation of the fusion algorithm. In the case of belief potentials, binary fusion is determined by

$$\begin{aligned} [\text{Fus}_C(\varphi_1, \varphi_2)(X)]_m &= \sum_{(X_1^{\uparrow D} \cap X_2^{\uparrow D})^{\downarrow C} = X} [\varphi_1(X_1)]_m [\varphi_2(X_2)]_m \\ &= \sum_{\substack{(X_1^{\uparrow D} \cap X_2^{\uparrow D})^{\downarrow C} = X \\ X_1 \in \text{FS}(\varphi_1), X_2 \in \text{FS}(\varphi_2)}} [\varphi_1(X_1)]_m [\varphi_2(X_2)]_m \end{aligned} \quad (10)$$

which is a simple consequence of Equations 8 and 9.

Another important remark is that normalization can be done either before or after combination, marginalization, or fusion. Formally, we can write $\nu(\varphi_1 \otimes \varphi_2) = \nu(\nu(\varphi_1) \otimes \nu(\varphi_2))$, $\nu(\varphi^{\downarrow C}) = \nu(\varphi)^{\downarrow C}$, and $\nu(\text{Fus}_C(\varphi_1, \varphi_2)) = \nu(\text{Fus}_C(\nu(\varphi_1), \nu(\varphi_2)))$, respectively. Normalization, if necessary, therefore can always be postponed to the end.

3. REPRESENTING FOCAL SETS

Belief potentials are completely determined by its focal sets and the corresponding masses. Consequently, a belief potential φ usually is represented by the set $\{(F_1, m_1), \dots, (F_k, m_k)\}$ of all pairs (F_i, m_i) with $F_i \in \text{FS}(\varphi)$ and $m_i = [\varphi(F_i)]_m$. Of course, the efficiency of computations is then strongly affected by the encoding of these focal sets. For that reason, we are particularly interested in encodings that allow computation of the main operations $X_1 \cap X_2$ (intersection), $X_1 = X_2$ (equality testing), $X^{\downarrow C}$ (projection), and $Y^{\uparrow D}$ (extension) for $X, X_1, X_2 \subseteq \Theta_D$, $Y \subseteq \Theta_C$, and $C \subseteq D$ as fast as possible. The most straightforward approach is to store focal sets by corresponding lists of configurations. This is of course not very sophisticated and will not be considered a possible candidate.

3.1. Binary Representation

The key for introducing a binary representation of focal sets is a global ordering of all the variables involved. For that purpose, let $V = \{x_1, \dots, x_g\}$ be the global set of all available variables. The ordering of the variables is implicitly determined by their indices. Without loss of generality, we suppose that every variable $x_i \in V$ has a set $\Theta_{x_i} = \{0, \dots, S_i - 1\}$ of possible values. Furthermore, let $D = \{x_{k_1}, \dots, x_{k_n}\} \subseteq V$ be a subset of variables of increasing indices k_i . Note that the set $\Theta_D = \{\mathbf{c}_0, \dots, \mathbf{c}_{S-1}\}$ contains exactly $S = \prod_{i=1}^n S_{k_i}$ configurations. For each configuration $\mathbf{c}_r = (r_1, \dots, r_n) \in \Theta_D$, the index r is unambiguously determined by the values r_1 and r_n and

^bNote that essentially the same technique is known under different names such as *bucket elimination*⁵ or, more generally, in the frameworks of *information algebras*^{12,13,15} and *valuation algebras*.²⁷

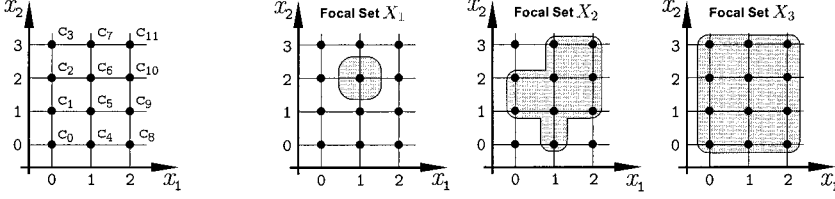


Figure 2. Representing sets of configurations $X \subseteq \Phi_D$.

$$r = \sum_{i=1}^n \left(r_i \prod_{j=i+1}^n S_{k_j} \right) \quad (11)$$

This way of enumerating the configurations is shown on the left side of Figure 2. If $X \subseteq \Theta_D$ is an arbitrary set of configurations, then a bit string $B_D(X) \stackrel{\text{def}}{=} \langle b_{S-1} \cdots b_0 \rangle$ with

$$b_i = \begin{cases} 1, & \text{if } \mathbf{c}_i \in X \\ 0, & \text{otherwise} \end{cases} \quad (12)$$

defines unequivocally a representation of the set X .^{17,33} Clearly, if $X_1, X_2 \subseteq \Theta_D$ are two sets of configurations, then $\text{logand}(B_D(X_1), B_D(X_2))$ denotes the bit string obtained from $B_D(X_1)$ and $B_D(X_2)$ by performing a bit-wise logical and (see Subsection 4.1). Of course, $\text{logand}(B_D(X_1), B_D(X_2))$ is the binary representation of $X_1 \cap X_2$ and is computed very efficiently on today's computers. Similarly, the equality $X_1 = X_2$ is tested easily by a simple comparison of the corresponding bit strings. However, the operations of projection $X \downarrow^C$ and extension $Y \uparrow^D$ are more expensive (see Section 4).

Example 2. Consider the sets $X_1, X_2, X_3 \subseteq \Theta_D$ of Figure 2 for $D = \{x_1, x_2\}$ and with $\Theta_{x_1} = \{0, 1, 2\}$ and $\Theta_{x_2} = \{0, 1, 2, 3\}$. By the method described in Equations 11 and 12, we get the following bit strings:

$$\begin{aligned} B_D(X_1) &= \langle 000001000000 \rangle, \\ B_D(X_2) &= \langle 111011110110 \rangle, \quad \text{and} \\ B_D(X_3) &= \langle 111111111111 \rangle \end{aligned}$$

The size of a bit string $B_D(X)$ is constantly S bits for every $X \subseteq \Theta_D$. As a consequence, the space complexity of encoding focal sets by bit strings is exponential with the number of variables in D . Thus, the applicability of this techniques is restricted to domains of moderate size. In practice, domain sizes of up to 12 to 15 binary variables work pretty well. In most cases, thanks to the technique of local computations, this is above the maximal induced domain size. However, in cases in which the sizes of the domains exceed this limit, other representations may become favorable.

3.2. Other Representations

The most promising alternatives are *logical* representations. If D is restricted to binary variables, then *propositional logic* is sufficient. The idea is to consider the variables $x_i \in D$ as propositions and to find a logical formula $\xi \in \mathcal{L}_D$ in which the set of models $M_D(\xi)$ corresponds to $X \subseteq \Theta_D$. Of course, there are many such formulas and the question is how to select one of them.

One possibility is to select either a *disjunctive normal form* (DNF) or a *conjunctive normal form* (CNF). In both cases, there is still a number of different DNFs or CNFs with the same set of models. This makes the equality test difficult (there is no polytime algorithm). Furthermore, intersection is expensive in the case of DNFs and projection is expensive for CNFs. An alternative is to work with *prime implicants* as a particular DNF or *prime implicates* as a particular CNF. In both cases, equality testing becomes easy, but the corresponding formulas are less succinct than arbitrary DNFs or CNFs.⁴ Furthermore, either intersection or projection remains difficult.

Another logical representation provides the technique of *ordered binary decision diagrams* (OBDD).³ They are interesting because polytime algorithm exists for all the necessary operations (intersection, equality testing, projection, and extension).⁴ OBDDs are very successful in the domain of formal verification, but so far, nobody has tried to use them within the Dempster-Shafer theory as a possible way of encoding focal sets. Despite possible benefits of using OBDDs, this article focuses on binary representation. A more comprehensive discussion of alternative representations can be found in Ref. 17.

4. PROJECTION AND EXTENSION

If focal sets are represented by bit strings, then projection and extension of such bit strings are the two crucial operations. The technique based on *bit masks* as described in Ref. 17 is only satisfactory for relatively small domains. In this section, using a few basic operations for bit strings, we propose a more efficient way of implementing projection and extension.

4.1. Bit String Operations

Let $\langle 0 \cdots 0b_{R-1} \cdots b_0 \rangle$ be a bit string of length S in which the $S - R$ left-most bits are all set to 0. In such a case, only the R right-most bits are significant. Thus, bit string $\langle b_{R-1} \cdots b_0 \rangle$ of length R is equivalent and may be used to represent the corresponding bit string of length $S \geq R$ in which the $S - R$ left-most bits are all set to 0. Consequently, every bit string of arbitrary length represents unambiguously a corresponding *infinite* bit string.

This point of view is very convenient and allows us to define the basic operations for bit strings without worrying about their respective lengths. We use \mathcal{B} to denote the set of all such infinite bit strings. Note that the bit string $\langle \rangle$ of length 0 represents $\langle \cdots 000 \rangle \in \mathcal{B}$ in which the bits are all set to 0. Consider now the following basic operations for arbitrary bit strings $B, B_1, \dots, B_m \in \mathcal{B}$:

- $\text{logand}(B_1, B_2)$: bit-wise logical and
- $\text{logior}(B_1, B_2)$: bit-wise logical inclusive or
- $\text{lsl}(B, n)$: logical shift to the left (n positions, fill the n right-most bits with 0's)
- $\text{lsr}(B, n)$: logical shift to the right (n positions)

All these basic operations are available and extremely efficient in today's micro-processors. Up to a certain length of the bit strings (typically 64 or 128 bits), they can be executed in constant time. In general, we have a linear time complexity $O(R)$ for bit strings of length R . As already mentioned in Subsection 3.1, $\text{logand}(B_1, B_2)$ can be used to compute the intersection $X_1 \cap X_2$ of two sets $X_1, X_2 \subseteq \Theta_D$ with $B_1 = B_D(X_1)$ and $B_2 = B_D(X_2)$. Similarly, $\text{logior}(B_1, B_2)$ determines the corresponding set union $X_1 \cup X_2$. With the aid of these basic operations, two additional bit string operations can be implemented, both of them playing an important role in the method presented in Subsection 4.2:

- $\text{extract}(B, n, \text{pos})$: extracts n bits from B , starting at position pos
- $\text{deposit}(B, n, \text{pos}, B')$: replaces n bits at position pos of B by the first n bits of B'

Both operations involve two basic operations (logand and lsr in the case of extract and logior and lsl in the case of deposit). Thus, up to a certain string length R , both extract and deposit can be done in constant time. However, in the general case, we have a linear time complexity $O(R)$ as shown previously.

4.2. Bit Block Shifting

Using *bit masks* as proposed in Ref. 17, the time for projecting or extending a bit string depends linearly on its length and thus exponentially on the size of the domain involved. Of course, this is quite unfavorable in practice, even if the maximal size of the domains is usually very limited in a framework of local computation. In the following, we present a more efficient approach that is based on a step-by-step procedure. In other words, the variables of $D \setminus C$ are *eliminated* or *adjoined* one after another.

For this purpose, consider first the special case where $C = D \setminus \{x_i\}$ for an arbitrary variable $x_i \in D$. If the variable x_i is at the s th position in $D = \{x_{k_1}, \dots, x_{k_n}\}$, then D can be divided into

$$D = \underbrace{\{x_{k_1}, \dots, x_{k_{s-1}}\}}_{L_D(x_i)} \underbrace{\{x_i, x_{k_{s+1}}, \dots, x_{k_n}\}}_{R_D(x_i)} \quad \text{with} \quad \begin{cases} u_D(x_i) \stackrel{\text{def}}{=} |\Theta_{L_D(x_i)}| \\ v_D(x_i) \stackrel{\text{def}}{=} |\Theta_{R_D(x_i)}| \end{cases}$$

First, consider the problem of projecting a set $X \subseteq \Theta_D$ to C . The pseudocode of the algorithm $\text{simple-projection}(B, x_i)$ is given below. The input parameters are the bit string $B = B_D(X)$ and the variable $x_i \in D$ to be eliminated. The idea of the algorithm is to regroup $|\Theta_{x_i}|$ blocks of the original bit string B by shifting its bits to a common position. The result is a new bit string from which the resulting bit blocks are extracted and successively deposited in another bit string R . Finally, the algorithm returns the bit string $R = B_C(X \downarrow^C)$.

Second, consider a set $Y \subseteq \Theta_C$ to be extended from $C = \{x_{k_1}, \dots, x_{k_{s-1}}, x_{k_{s+1}}, \dots, x_{k_n}\}$ to $D = \{x_{k_1}, \dots, x_{k_{s-1}}, x_i, x_{k_{s+1}}, \dots, x_{k_n}\}$. Again, we use $u_D(x_i)$ and $v_D(x_i)$ to denote the number of configurations of the corresponding subdomains. The following pseudocode describes the algorithm `simple-extension(B, x_i)` with $B = B_C(Y)$ and $x_i \in D$ as input parameters. Basically, the idea of the algorithm is to reverse projection.

Algorithm: `simple-projection(B, x_i)`;

```
[01]  $R := \langle \rangle$ ;
[02]  $u := u_D(x_i)$ ;  $v := v_D(x_i)$ ;
[03] For  $k$  From 1 To  $|\Theta_{x_i}| - 1$  Do
[04]    $B := \text{logior}(B, \text{lsl}(B, v))$ ;
[05] Next;
[06] For  $k$  From 0 To  $u - 1$  Do
[07]    $E := \text{extract}(B, v, k|\Theta_{x_i}|v)$ ;
[08]    $R := \text{deposit}(R, v, kv, E)$ ;
[09] Next;
[10] Return  $R$ ;
```

Algorithm: `simple-extension(B, x_i)`;

```
[01]  $R := \langle \rangle$ ;
[02]  $u := u_D(x_i)$ ;  $v := v_D(x_i)$ ;
[03] For  $k$  From 0 To  $u - 1$  Do
[04]    $E := \text{extract}(B, v, kv)$ ;
[05]    $R := \text{deposit}(R, v, k|\Theta_{x_i}|v, E)$ ;
[06] Next;
[07] For  $k$  From 1 To  $|\Theta_{x_i}| - 1$  Do
[08]    $R := \text{logior}(R, \text{lsl}(R, v))$ ;
[09] Next;
[10] Return  $R$ ;
```

The execution time of the foregoing algorithms depends mainly on the total number $u_D(x_i) + |\Theta_{x_i}| - 1$ of iterative steps. Because $u_D(x_i)$ depends exponentially on $|L_D(x_i)| = s - 1$ and thus on the position s of the variable x_i in D , we have an overall time complexity of $O(2^{s-1})$. Thus, using the foregoing algorithms, projection and extension are particularly efficient if the variable x_i has a small index relative to D , but in the worst case, i.e., if $s = |D|$, we have the same complexity as the bit mask method described in Ref. 17.

Example 3. Consider the set $X_1 \subseteq \Theta_D$ of Figure 2 for $D = \{x_1, x_2\}$ and with $\Theta_{x_1} = \{0, 1, 2\}$ and $\Theta_{x_2} = \{0, 1, 2, 3\}$. The bit string for X_1 is $B = B_D(X_1) = \langle 000001000000 \rangle$. Suppose that x_1 is the variable to be eliminated. We have then $u = u_D(x_1) = 1$ (because x_1 has the smallest index in D and thus $L_D(x_1) = \emptyset$), $v = v_D(x_1) = 4$ (because $R_D(x_1) = \{x_2\}$ and $|\Theta_{x_2}| = 4$), and $|\Theta_{x_1}| = 3$. Thus, the first loop in `simple-projection` iterates from $k = 1$ to $k = 3 - 1 = 2$ and changes the bit string B as follows:

$$B = \langle 000001000000 \rangle$$

$$\text{lsl}(B, 4) = \langle 000000000100 \rangle$$

$$B := \text{logior}(B, \text{lsl}(B, 4)) = \langle 000001000100 \rangle$$

$$\text{lsl}(B, 4) = \langle 000000000100 \rangle$$

$$B := \text{logior}(B, \text{lsl}(B, 4)) = \langle 000001000100 \rangle$$

The second loop iterates just once (from $k = 0$ to $k = 1 - 1 = 0$). It simply extracts the $v = 6$ lowest bits from B and deposits them into R . The resulting bit string $R = \langle 0100 \rangle$ is the corresponding representation of $X_1 \downarrow_{\{x_2\}}$ with respect to the new domain $\{x_2\}$ (see Figure 2).

Example 4. Consider the resulting bit string $R = \langle 0100 \rangle$ obtained from the previous example and suppose R is taken as input value B for simple-extension. Furthermore, let x_1 be the new variable to be introduced. We have again $u = u_D(x_1) = 1$, $v = v_D(x_1) = 4$, and $|\Theta_{x_1}| = 3$. Thus, the first iteration is a single step in which the four bits of B are simply copied into R . Then, during the second iteration, R changes as follows:

$$\begin{aligned} R &= \langle 000000001000 \rangle \\ \text{Isl}(R, 4) &= \langle 000001000000 \rangle \\ R := \text{logior}(R, \text{Isl}(R, 4)) &= \langle 000001000100 \rangle \\ \text{Isl}(R, 4) &= \langle 010001000000 \rangle \\ R := \text{logior}(R, \text{Isl}(R, 4)) &= \langle 010001000100 \rangle \end{aligned}$$

The resulting bit string $R = \langle 010001000100 \rangle$ is the corresponding representation of the set $(X_1^{\downarrow \{x_2\}})^{\uparrow D}$ with respect to the original domain $D = \{x_1, x_2\}$. It contains three configurations \mathbf{c}_2 , \mathbf{c}_6 , and \mathbf{c}_{10} (see Figure 2).

Now, let us look at the general case where C is an arbitrary subset of D . If $D \setminus C = \{x_{i_1}, \dots, x_{i_m}\}$ is the set of variables to be eliminated or adjoined, then projection and extension are sequential procedures of the form

$$\begin{aligned} X &\Rightarrow X^{\downarrow D - \{x_{i_1}\}} \Rightarrow X^{\downarrow D - \{x_{i_1}, x_{i_2}\}} \Rightarrow \dots \Rightarrow X^{\downarrow D - \{x_{i_1}, \dots, x_{i_m}\}} = X^{\downarrow C} \\ Y &\Rightarrow Y^{\uparrow C \cup \{x_{i_1}\}} \Rightarrow Y^{\uparrow C \cup \{x_{i_1}, x_{i_2}\}} \Rightarrow \dots \Rightarrow Y^{\uparrow C \cup \{x_{i_1}, \dots, x_{i_m}\}} = Y^{\uparrow D} \end{aligned}$$

This is true because projecting and extending sets of configurations are transitive. Note that any ordering of the variables in $D \setminus C$ is possible. However, because the efficiency of the foregoing algorithms simple-projection and simple-extension depends on the position s of the variable x_i in D , we propose to select iteratively the variable $x_i \in D \setminus C$ in which the position s in D is minimal for projection, and, conversely, the variable $x_i \in D \setminus C$ in which the position s in D is maximal for extension. If s_{\max} denotes the maximal position of the m variables of $D \setminus C$ in D , then we get an overall time complexity of $O(m2^{s_{\max} - m})$ for both projection and extension. In the average case, this is much better than the complexity resulting from the bit mask method described in Ref. 17. Regardless of these heuristics, the two algorithms can be described as follows:

Algorithm: projection(B, c);

[01] For Each $x_i \in D \setminus C$ Do

[02] $B := \text{simple-projection}(B, x_i)$;

[03] Next;

[04] Return B ;

Algorithm: extension(B, D);

[01] For Each $x_i \in D \setminus C$ Do

[02] $B := \text{simple-extension}(B, x_i)$;

[03] Next;

[04] Return B ;

Note that a group of variables $G \subseteq D \setminus C$ in which the elements are next to each other in D can be treated as one single variable x_G with $\Theta_{x_G} = \Theta_G$. This may further increase the efficiency of the foregoing procedures, especially when the variables of G have high indices relative to D .

4.3. Memoizing

In the case of the fusion operator, an important improvement of the methods presented in the Subsection 4.2 is to *memoize* projection. A memoized function caches its return values. Later, if the function is called with the same arguments, it returns the cached value instead of recomputing the same return value. Memoizing usually is implemented with the aid of hash tables.

In our case, memoizing projection can be installed generally for both simple-projection and projection. However, the main benefit results from installing memoizing for projection in the case of the fusion operator where many intersections of focal sets are equal) (see Subsection 5.4). Note that the cached return values (hash table) of the memoized version of projection should be cleared after completing fusion.

4.4. Quasi-Projection

Using the block shifting method of Subsection 4.2, the most expensive part of projecting a bit string B to a smaller domain is extracting and rearranging the relevant bit blocks (see Algorithm simple-projection, [06]–[09]). Let us now introduce a new operation for which extracting and rearranging bit blocks are not necessary. If $X \subseteq \Theta_D$ is a set of configurations and $C \subseteq D$ the new domain, then

$$X \uparrow^C \stackrel{\text{def}}{=} (X \downarrow^C) \uparrow^D \quad (13)$$

defines the *quasi-projection* of X to C . This simple operation is illustrated by the example in Figure 1. Note that the elements of $X \uparrow^C$ are configurations relative to D .

Consider now the definitions of marginalization in Equation 9 and fusion in Equation 10. In both cases, it is possible to rewrite the restriction that determines the respective sum. We can write $Y \uparrow^C = X \uparrow^D$ instead of $Y \downarrow^C = X$ for marginalization and $(X_1 \uparrow^D \cap X_2 \uparrow^D) \uparrow^C = X \uparrow^D$ instead of $(X_1 \uparrow^D \cap X_2 \uparrow^D) \downarrow^C = X$ for fusion. Thus, it is possible to sum up over equal quasi-projections instead of equal projections. Computing actual projections is then only necessary for each of the resulting focal sets (See Subsections 5.3 and 5.4 for more details).

As in the previous subsection, the idea for implementing quasi projection is to treat the variables $x_i \in D \setminus C$ one after another. We use $M_D(x_i)$ to denote a bit mask that determines the $u_D(x_i)$ relevant bit blocks of length $v_D(x_i)$ for the variable x_i (see Algorithm simple-projection, [06]–[09]).

Algorithm: simple-qprojection(B, x_i);

```
[01]  $v := v_D(x_i)$ ;
[02] For  $k$  From 1 To  $|\Theta_{x_i}| - 1$  Do
[03]    $B := \text{logior}(B, \text{lsr}(B, v))$ ;
[04] Next;
[05]  $B := \text{logand}(B, M_D(x_i))$ ;
[06] For  $k$  From 1 To  $|\Theta_{x_i}| - 1$  Do
[07]    $B := \text{logior}(B, \text{lsl}(B, v))$ ;
[08] Next;
[09] Return  $B$ ;
```

Algorithm: qprojection(B, C);

```
[01] For Each  $x_i \in D \setminus C$  Do
[02]    $B := \text{simple qprojection}(B, x_i)$ ;
[03] Next;
[04] Return  $B$ ;
```

The first few lines of simple-qprojection are similar to simple-projection. In line [05], the bit mask $M_D(x_i)$ is then used to set all irrelevant bits to zero. Finally, in a similar way as in simple-extension, the relevant bit blocks are repeatedly duplicated.^c

Example 5. Consider again the set $X_1 \subseteq \Theta_D$ of Figure 2 for $D = \{x_1, x_2\}$ and with $\Theta_{x_1} = \{0, 1, 2\}$ and $\Theta_{x_2} = \{0, 1, 2, 3\}$. The bit string $B = B_D(X_1) = \langle 000001000000 \rangle$ and x_1 are taken as the input parameters for simple-qprojection. Again, we have $v = v_D(x_1) = 4$ and $|\Theta_{x_1}| = 3$ (see Example 3). Clearly, the first iteration alters B in the same way as in Example 3, thus producing $B = \langle 000001000100 \rangle$. Then, at step [05] of the algorithm, we get

$$B := \text{logand}(B, M_D(x_1)) = \langle 000000000100 \rangle$$

where $M_D(x_1) = \langle 000000001111 \rangle$ is the corresponding mask for x_1 . This is then the input parameter for the second loop that changes B as follows:

$$B = \langle 000000000100 \rangle$$

$$\text{lsl}(B, 4) = \langle 000001000000 \rangle$$

$$B := \text{logior}(B, \text{lsl}(B, 4)) = \langle 000001000100 \rangle$$

$$\text{lsl}(B, 4) = \langle 010001000000 \rangle$$

$$B := \text{logior}(B, \text{lsl}(B, 4)) = \langle 010001000100 \rangle$$

Clearly, the resulting bit string $B = \langle 010001000100 \rangle$ is the binary representation of the quasi projection $X_1^{\downarrow\{x_2\}}$ (see Figure 2).

Note that the efficiency of the foregoing iterative procedure depends only on $|\Theta_{x_i}|$ but not on $u_D(x_i)$. Thus, the positions of the variables to be eliminated are no longer relevant. This is the reason why we consider quasi-projection as an important tool that significantly reduces the time for marginalization and fusion. This remark will be underlined by the empirical tests in Subsection 6.2. Note that in the case of the fusion operator, it is again important to install memoizing for qprojection.

5. COMBINATION, MARGINALIZATION, AND FUSION

Let us now turn our attention to the main higher-level operations for belief potentials: combination, marginalization, and, alternatively, fusion. A common issue of all these operations is regrouping.

5.1. Regrouping

As already mentioned in Section 3, a belief potential $\varphi \in \Phi_D$ usually is represented by the set $\mathcal{F}_\varphi = \{(F_1, m_1), \dots, (F_k, m_k)\}$ of pairs (F_i, m_i) with

^cDuplicating the relevant bits can also be omitted. The important point is to have a function $f(B)$ such that $B^{\downarrow C} = B'^{\downarrow C} \Leftrightarrow f(B) = f(B')$. Quasi-projection as defined previously is one possibility, but the intermediate result at line [05] of simple-qprojection would be another one. Note that in this way, the efficiency of the foregoing procedure would approximately be doubled. All the tests in Subsection 6.2 are based on this optimized version of quasi-projection.

$F_i \in FS(\varphi)$ and $m_i = [\varphi(F_i)]_m$. More generally, let $\mathcal{F} = \{(F_1, m_1), \dots, (F_k, m_k)\}$ be an arbitrary set of pairs (F_i, m_i) with $F_i \subseteq \Theta_D$ and $0 \leq m_i \leq 1$ for $1 \leq i \leq k$ and such that $\sum_{i=1}^k m_i = 1$. Note that the sets F_i are not necessarily distinct. Such sets arise as intermediate results during combination, marginalization, and fusion. The problem then is to *regroup* equal sets and to *sum up* their respective values. The following algorithm shows a general solution:

Algorithm: regroup(\mathcal{F});
 [01] $\mathcal{R} := \emptyset$;
 [02] For Each $(F, m) \in \mathcal{F}$ Do
 [03] If update(\mathcal{R}, F, m) = *false* Then $\mathcal{R} := \mathcal{R} \cup \{(F, m)\}$;
 [04] Next;
 [05] Return \mathcal{R} ;

The foregoing procedure iterates through all the pairs $(F, m) \in \mathcal{F}$. At each individual step, update(\mathcal{R}, F, m) tries to find another pair $(F', m') \in \mathcal{R}$ with $F' = F$. If such a pair exists, then m' is replaced by $m' + m$. Otherwise, update(\mathcal{R}, F, m) returns *false* and (F, m) is adjoined to \mathcal{R} .

If the set of pairs is represented by simple lists, then the complexity of the foregoing procedure is $O(k^2)$. A slightly better method is to consider bit strings $B_D(F)$ as integers and to use *ordered* instead of ordinary lists. However, the best average lookup times are obtained from using either *balanced binary trees* (in particular *AVL* or *red-black trees*) or *hash tables*.¹¹ Then, the resulting complexity of the foregoing procedure is $O(k \log k)$ for balanced trees and $O(k^2/s)$ for hash tables of size $s \leq k$. The empirical tests in Section 6.2 will show the importance of using such techniques.

5.2. Combination

From a computational point of view, combining two belief potentials φ_1 and φ_2 by Dempster's rule of combination is done in three steps. First, all the focal sets $F_1 \in FS(\varphi_1)$ and $F_2 \in FS(\varphi_2)$ are extended to $D = D_1 \cup D_2$. Second, every extended focal set $F_i^{\uparrow D}$ is intersected with every extended focal set $F_j^{\uparrow D}$ and their respective masses $[\varphi_1(F_1)]_m$ and $[\varphi_2(F_2)]_m$ are multiplied. Finally, equal intersections are regrouped and their respective masses $[\varphi_1(F_1)]_m[\varphi_2(F_2)]_m$ are summed up.

Algorithm: combination($\mathcal{F}_1, \mathcal{F}_2$);
 [01] For Each $(F, m) \in \mathcal{F}_1 \cup \mathcal{F}_2$ Do
 [02] $F := F^{\uparrow D}$;
 [03] Next;
 [04] $\mathcal{R} := \emptyset$;
 [05] For Each $(F_1, m_1) \in \mathcal{F}_1$ Do
 [06] For Each $(F_2, m_2) \in \mathcal{F}_2$ Do
 [07] $\mathcal{R} := \mathcal{R} \cup \{(F_1 \cap F_2, m_1 m_2)\}$;
 [08] Next;
 [09] Next;
 [10] $\mathcal{R} :=$ regroup(\mathcal{R});
 [11] Return \mathcal{R} ;

Of course, by extending line [07] of combination according to line [03] of regroup, it is possible to incrementally regroup equal intersections during the procedure. As a consequence, line [10] can then be omitted.

5.3. Marginalization

The implementation of marginalization depends on whether it is based on projection or quasi-projection (see Subsection 4.4). Therefore, by the pseudocode given in the following, we propose two variants of the marginalization procedure:

Algorithm: marginalization(\mathcal{F} , C);

[01] For Each $(F, m) \in \mathcal{F}$ Do

[02] $F := F \downarrow^C$;

[03] Next;

[04] $\mathcal{F} := \text{regroup}(\mathcal{F})$;

[05] Return \mathcal{F} ;

Algorithm: qmarginalization(\mathcal{F} , C);

[01] For Each $(F, m) \in \mathcal{F}$ Do

[02] $F := F \uparrow^C$;

[03] Next;

[04] $\mathcal{F} := \text{regroup}(\mathcal{F})$;

[05] For Each $(F, m) \in \mathcal{F}$ Do

[06] $F := F \downarrow^C$;

[07] Next;

[08] Return \mathcal{F} ;

In both cases, the main issue is to iterate through the focal sets $F \in \text{FS}(\varphi)$ and to regroup the corresponding results $F \downarrow^C$ and $F \uparrow^C$, respectively. In the case of qmarginalization, another iteration is then necessary in order to actually project the resulting focal sets to the new domain. Note that equal sets again may be regrouped incrementally.

5.4. Fusion

Fusion is similar to combination except that every intersection $F_1 \cap F_2$ of two focal sets $F_1 \in \mathcal{F}_1$ and $F_2 \in \mathcal{F}_2$ is immediately projected to the new domain C . Again, we propose two variants fusion and qfusion, depending on whether one uses projection or quasi-projection.

Algorithm: fusion(\mathcal{F}_1 , \mathcal{F}_2 , c);

[01] For Each $(F, m) \in \mathcal{F}_1 \cup \mathcal{F}_2$ Do

[02] $F := F \uparrow^D$;

[03] Next;

[04] $\mathcal{R} := \emptyset$;

[05] For Each $(F_1, m_1) \in \mathcal{F}_1$ Do

[06] For Each $(F_2, m_2) \in \mathcal{F}_2$ Do

[07] $\mathcal{R} := \mathcal{R} \cup \{((F_1 \cap F_2) \downarrow^C, m_1 m_2)\}$;

[08] Next;

[09] Next;

[10] $\mathcal{R} := \text{regroup}(\mathcal{R})$;

[11] Return \mathcal{R} ;

Algorithm: qfusion(\mathcal{F}_1 , \mathcal{F}_2 , C);

[01] For Each $(F, m) \in \mathcal{F}_1 \cup \mathcal{F}_2$ Do

[02] $F := F \uparrow^D$;

[03] Next;

[04] $\mathcal{R} := \emptyset$;

[05] For Each $(F_1, m_1) \in \mathcal{F}_1$ Do

[06] For Each $(F_2, m_2) \in \mathcal{F}_2$ Do

[07] $\mathcal{R} := \mathcal{R} \cup \{((F_1 \cap F_2) \uparrow^C, m_1 m_2)\}$;

[08] Next;

[09] Next;

[10] $\mathcal{R} := \text{regroup}(\mathcal{R})$;

[11] For Each $(F, m) \in \mathcal{R}$ Do

[12] $F := F \downarrow^C$;

[13] Next;

[14] Return \mathcal{R} ;

Again, note that by extending line [07] and removing line [10] of both fusion and qfusion, regrouping may also be done incrementally.

6. ARCHITECTURES AND EXPERIMENTAL RESULT

This section defines a test bed that allows a comparison of different implementation variants. We restrict the discussion on the problem of marginalizing the combination of two belief potentials $\varphi_1 \in \Phi_{D_1}$ and $\varphi_2 \in \Phi_{D_2}$ to a domain $C \subseteq D_1 \cup D_2$. This corresponds to $\text{Fus}_C(\varphi_1, \varphi_2)$ and is the basic computational step in Shenoy’s general approach to local computation using binary join trees (join trees are called *binary* when every node of the tree has at most three neighbors).²⁶ All the tests were taken on the basis of the binary representation of focal sets as proposed in Subsection 3.1. Our empirical investigation includes a total number of 24 implementation variants.

6.1. Architectures

To categorize the different implementation variants, we first distinguish three different top-level implementation architectures:

- (A1) *Classical Method*. Using the methods of Subsections 5.2 and 5.3, the two potentials φ_1 and φ_2 are combined, and the combined potential $\varphi_1 \otimes \varphi_2$ is marginalized to the new domain C .
- (A2) *Stepwise Marginalization*. Again, the two potentials φ_1 and φ_2 are combined by the method of Subsection 5.2, but the combined potential $\varphi_1 \otimes \varphi_2$ is marginalized in a step-by-step procedure to the new domain C , i.e., the variables of $D \setminus C$ are eliminated one after another.
- (A3) *Fusion*. Combination and marginalization is replaced by fusion as defined by Equation 10 and Subsection 5.4. We distinguish two alternatives:
 - (a) Without memoizing;
 - (b) With memoized projection as explained in Subsection 4.3 (using hash tables).

For each of these architectures, it is possible to define several alternatives, depending on whether regrouping is done with the aid of simple lists, balanced binary trees, or hash tables (see Subsection 5.1) and whether marginalization and fusion are implemented with quasi projection or not (see Subsections 4.4, 5.3, and 5.4).

6.2. Experimental Results

Our test bed consists of two belief potentials taken from the (binary) join tree of an example with 632 binary variables and 1101 initial belief potentials. During the propagation through the join tree, the sizes of the potentials involved increase steadily. Interestingly, more than 95% of the total propagation time is spent in less than 3% of the nodes involved. This indicates that attempts to improve the efficiency of belief function computations must focus on large belief potentials

Table 1. Characteristics of φ_1 , φ_2 , $\varphi_1 \otimes \varphi_2$, and $(\varphi_1 \otimes \varphi_2)^{\downarrow C}$.

	$ d(\varphi) $	$d(\varphi)$	$ \text{FS}(\varphi) $
φ_1	8	$\{a, b, c, d, e, g, i, j\}$	1,862
φ_2	8	$\{d, e, f, g, h, i, j, k\}$	1,135
$\varphi_1 \otimes \varphi_2$	11	$\{a, b, c, d, e, f, g, h, i, j, k\}$	154,581
$(\varphi_1 \otimes \varphi_2)^{\downarrow C}$	6	$\{a, b, c, f, h, k\}$	160

with possibly several thousand focal sets. The selected potentials φ_1 and φ_2 are the two incoming messages of a node selected arbitrarily among these crucial nodes. Table 1 shows the characteristics of φ_1 , φ_2 , its combination $\varphi_1 \otimes \varphi_2$, and the corresponding marginal $(\varphi_1 \otimes \varphi_2)^{\downarrow C}$.

Note that combining φ_1 and φ_2 involves $1,862 \cdot 1,135 = 2,113,370$ intersections of focal sets. The length of corresponding bit strings is constantly $2^{11} = 2048$ bits. After regrouping, the number of focal sets drops to 154,581. Finally, only 160 focal sets remain after marginalization. Such a tremendous reduction of the size is very typical for cases where several variables are eliminated. It indicates the importance of efficient regrouping. Table 2 shows the necessary times for computing $(\varphi_1 \otimes \varphi_2)^{\downarrow C}$ using 24 different implementation variants. The same tests were also repeated for the bit mask method as proposed in Ref. 17, but the corresponding results are not competitive. The experimental framework was implemented in MCL 4.3 (Macintosh Common Lisp), and all the tests were taken on the same 400-MHz Power Mac G3 with 768 MByte RAM. Let us discuss some observations:

- For A1 and A2, regrouping on the basis of simple lists is extremely slow (more than 18 hours!). A tremendous improvement results from using either AVL trees or hash tables.
- Regrouping with hash tables is up to 30% faster than regrouping with AVL trees (provided that the hash tables are large enough).
- Quasi-projection provides another significant improvement, especially for A1, A3a, and A3b.
- Fusion without memoizing is relatively slow.
- The best results are observed for A1, A2, and A3b using hash tables for regrouping and quasi-projection. No significant difference is observed among them.

Note that solely combining the two belief potentials requires 41.1 seconds in the case of hash tables and 70.5 seconds in the case of AVL trees. Therefore,

Table 2. Times for computing $(\varphi_1 \otimes \varphi_2)^{\downarrow C}$.

	Projection			Quasi-projection		
	Simple lists	AVL trees	Hash tables	Simple lists	AVL trees	Hash tables
A1	66,549.0 sec	144.2 sec	109.7 sec	66,494.2 sec	86.1 sec	50.6 sec
A2	68,136.4 sec	96.5 sec	59.2 sec	68,559.9 sec	87.9 sec	51.4 sec
A3a	909.5 sec	912.3 sec	898.9 sec	151.4 sec	145.4 sec	130.4 sec
A3b	114.4 sec	113.4 sec	112.2 sec	53.8 sec	52.5 sec	51.0 sec

approximately 80% of the time required for the complete procedure is needed for combining the two belief potentials (more than 2 million intersections of focal sets plus regrouping), whereas marginalization takes up only 20% of the total time. In contrast, without quasi-projection, marginalization and combination are more or less equally expensive. This indicates that implementing marginalization or fusion on the basis of quasi-projection apparently is close to the optimum.

7. CONCLUSIONS

This article provides a comprehensive study of different implementation aspects for Dempster-Shafer belief functions. Several conclusions may be drawn from the results. First, efficient regrouping is crucial and, preferably, is implemented with the aid of hash tables. Second, projecting and extending sets of configurations as proposed in Subsection 4.2 is much more efficient than the method presented in Ref. 17. Third, a significant improvement results from replacing projection by quasi projection as defined in Subsection 4.4. Furthermore, fusion is only competitive if memoizing is installed for (quasi) projection; and, finally, there are at least three different implementation options for which equally good experimental results are obtained.

The methods presented here are all based on a binary representation of focal sets. This, future work may focus on the investigation of other representations. Of particular interest are ordered binary decision diagrams as suggested in Subsection 3.2. However, because even the best representation and the most sophisticated algorithms do not help to overcome the computational limitations of Dempster-Shafer belief functions, one should consider the contribution of this study only in combination with corresponding approximation methods.

Acknowledgments

We would especially like to thank the anonymous reviewer for providing numerous suggestions and constructive comments on various aspects of this study. Research supported by Scholarship 8220-061232 and Grant 2000-061454.00 of the Swiss National Science Foundation.

References

1. Dempster AP, Kong A. Uncertain evidence and artificial analysis. *J Stat Plann Infer* 1988;20:355–368.
2. Bauer M. Approximations for decision making in the Dempster-Shafer theory of evidence. In: Horvitz E, Jensen F, editors. *Proc 12th Conf on Uncertainty in Artificial Intelligence (UAI-96)*. San Mateo, CA: Morgan Kaufmann Publishers; 1996. pp 73–80.
3. Bryant RE. Graph-based algorithms for boolean function manipulation. *IEEE Trans Comput* 1986;C-35(8):677–692.
4. Darwiche A, Marquis P. A perspective on knowledge compilation. In: *Proc 17th Int Joint Conf on Artificial Intelligence (IJCAI'01)*. Seattle, August 4–10, Morgan Kaufmann, 2001. pp 175–182.
5. Dechter R. Bucket elimination: A unifying framework for reasoning. *Artif Intell* 1999; 113(1–2):41–85.

6. Dempster A. Upper and lower probabilities induced by a multivalued mapping. *Ann Math Stat* 1967;38:325–339.
7. Denoeux T. Inner and outer clustering approximations of belief structures. In: *Proc 8th Int Conf (IPMU'00)*. Madrid, Spain, 2000. pp 125–132.
8. Haenni R, Kohlas J, Lehmann N. Probabilistic argumentation systems. In: Kohlas J, Moral S, editors. *Handbook of defeasible reasoning and uncertainty management systems, Vol 5: Algorithms for uncertainty and defeasible reasoning*. Dordrecht Kluwer Academic Publishers; pp 221–288, 2001.
9. Haenni R, Lehmann N. Resource-bounded and anytime approximation of belief function computations, *Int J Approximate Reasoning* 2002; 32(1–2):103–154.
10. Harmanec D. Faithful approximations of belief functions. In: Laskey KB, Prade H, editors. *Proc 15th Conf on Uncertainty in Artificial Intelligence (UAI-99)*. San Mateo, CA: Morgan Kaufmann Publishers; 1999. pp 271–278.
11. Knuth DE. *The art of computer programming, Sorting and searching, Vol 3, Series in computer science and information processing*. Reading, MA: Addison-Wesley; 1973.
12. Kohlas J. *Computational theory for information systems*. Technical Report 97-07, University of Fribourg, Institute of Informatics, 1997.
13. Kohlas J, Haenni R, Moral S. Propositional information systems. *J Logic Computat* 1999;9(5):651–681.
14. Kohlas J, Monney PA. A mathematical theory of hints. An approach to the Dempster-Shafer theory of evidence, *Lecture notes in economics and mathematical systems, Vol 425*. Berlin: Springer; 1995.
15. Kohlas J, Stärk R. Information algebras and information systems. Technical Report 96-14, University of Fribourg, Institute of Informatics, 1996.
16. Lauritzen S, Spiegelhalter DJ. Local computations with probabilities on graphical structures and their application to expert systems. *J R Stat Soc* 1988;50(2):157–224.
17. Lehmann N. *Argumentation systems and belief functions*. PhD thesis, University of Fribourg, Switzerland, 2001.
18. Lehmann N, Haenni R. An alternative to outward propagation for Dempster-Shafer belief functions. In: Hunter A, Parsons S, editor. *Symbolic and quantitative approaches to reasoning and uncertainty*. Berlin: Springer; 1998. pp 255–267.
19. Lowrance H, Garvey T, Strat T. A framework for evidential-reasoning systems. In: Shafer G, Pearl J (editors): *Uncertain Reasoning*, Morgan Kaufmann, 1990, pp 611–618.
20. Orponen P. Dempster's rule of combination is #P-complete. *Artif Intell* 1990;44:245–253.
21. Pearl J. *Probabilistic reasoning in intelligent systems*. San Mateo, CA: Morgan Kaufmann; 1988.
22. Shafer G. *The mathematical theory of evidence*. Princeton, NJ: Princeton University Press; 1976.
23. Shafer G. An axiomatic study of computation in hypertrees. Working Paper 232, School of Business, The University of Kansas, 1991.
24. Shafer G, Shenoy P. Axioms for probability and belief function propagation. In: Shafer G, Pearl J, editors. *Readings in uncertain reasoning*. San Mateo, CA: Morgan Kaufmann; 1990. pp 575–610.
25. Shenoy PP. Valuation-based systems: A framework for managing uncertainty in expert systems. In: Zadeh LA, Kacprzyk J, editors. *Fuzzy logic for the management of uncertainty*. New York: John Wiley and Sons; 1992. pp 83–104.
26. Shenoy PP. Binary join trees for computing marginals in the Shenoy-Shafer architecture. *Int J Approx Reason* 1997;17(2–3):239–263.
27. Shenoy PP, Kohlas J. Computation in valuation algebras. In: Gabbay D, Smets Ph, editors. *Handbook of defeasible reasoning and uncertainty management systems, Algorithms for*

- Uncertainty and Defeasible Reasoning, Vol 5. Dordrecht/New York/Norwell/London: Kluwer Academic Publishers; 2000. pp 5–39.
28. Shenoy PP, Shafer G. Propagating belief functions with local computations. *IEEE Expert*, 1986;1(3):43–52.
 29. Smets Ph. Belief functions. In: Dubois D, Smets Ph, Mamdani A, Prade H, editors. *Nonstandard logics for automated reasoning*. New York/London/Orlando, FL: Academic Press; 1988. pp 253–286.
 30. Smets Ph, Kennes R. The transferable belief model. *Artif Intell* 1994;66:191–234.
 31. Tessem B. Approximations for efficient computation in the theory of evidence. *Artif Intell* 1993;61(2):315–329.
 32. Voorbraak F. A computationally efficient approximation of Dempster-Shafer theory. *Int J Man Mach Stud* 1989;30(5):525–536.
 33. Xu H, Kennes R. Steps toward efficient implementation of Dempster-Shafer theory. In: Yager RR, Fedrizzi M, Kacprzyk J, editors. *Advances in the Dempster-Shafer theory of evidence*. New York: John Wiley & Sons; 1994. pp 153–174.