Marginalization without Summation
Exploiting Determinism in Factor Algebra

Sander Evers, Peter J.F. Lucas

Institute for Computer and Information Sciences
Radboud University Nijmegen
s.evers@cs.ru.nl, peterl@cs.ru.nl

Abstract. It is known that solving an exact inference problem on a discrete Bayesian network with many deterministic nodes can be far cheaper than what would be expected based on its treewidth. In this article, we introduce a novel technique for this: to the operations of factor multiplication and factor summation that form the basis of many inference algorithms, we add *factor indexing*. We integrate this operation into variable elimination, and extend the minweight heuristic accordingly. A preliminary empirical evaluation gives promising results.

Keywords: Bayesian networks, exact inference, factor algebra, deterministic variables

1 Introduction

In general, exact inference on a Bayesian network with discrete variables is known to take $O(d^w)$ time, where d is the domain size (assuming it is the same for each variable) and w is the treewidth of the network's moral graph[4]. In the canonical technique for exact inference, *variable elimination*[11], this constraint manifests itself as the minimal size of the largest factor that is created during the execution of the algorithm; implemented as a multidimensional array, it has w dimensions and d entries per dimension.

When a network contains deterministic nodes, inference can be much faster. One example where this can be seen is the approach of Chavira and Darwiche[1], in which a Bayesian network is transformed into a logical theory, and inference is performed by counting the models of this theory. These models should be consistent with the constraints imposed by the deterministic nodes. A good model counting algorithm can use these constraints effectively to prune the model search space. This approach, however, is quite remote from other inference algorithms in that it does not compute per-variable probabilities 'in bulk' by multiplication and summation of factors.

An approach by Larkin and Dechter[8] does use these factor operations. Here, a factor is implemented not as an array (with an entry for each possible variable assignment), but as a list of variable assignments that are nonzero (sometimes called a *sparse array*). The length of this list can be much smaller than the size of the array, but the overhead for multiplying and marginalizing factors is larger,

because the list has to be searched for values (possibly using a hash table). With this alternative implementation of factors, ordinary variable elimination can be performed.

The inference approach presented in this article is also based on the familiar factor operations. Firstly, we use a cheap representation of factors for deterministic variables. More importantly, we introduce a new marginalization method for these variables, which requires no summation and can therefore be much faster. We apply this method in a variable elimination algorithm, and propose an extended minweight heuristic informed by this method. However, because the marginalization method is formulated as a rewrite rule for factor expressions, its potential use is not limited to variable elimination, but can be extended to all inference algorithms that use factor operations.

The remainder of the article has the following outline. Sect. 2 summarizes the formal preliminaries for inference on Bayesian networks. In Sect. 3, we review variable elimination, with an emphasis on the use of factor algebra. Our main contribution, *factor indexing*, is presented in Sect. 4, followed by an empirical evaluation in Sect. 5. In Sect. 6, we conclude and propose future work.

2 Formal Preliminaries

We consider Bayesian networks over a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of n discrete variables; each V_i has a finite domain $\mathrm{dom}(V_i)$. Formally, an *instantiation* $\mathbf{v} = \{V_1{=}v_1, \ldots, V_n{=}v_n\}$ of these variables is a function that maps each V_i to a value $v_i \in \mathrm{dom}(V_i)$, often also called a *state*. With a little abuse of notation, we write $\mathbf{v}{\in}\mathbf{V}$ to let a variable $\mathbf{v}$ range over the possible instantiations of $\mathbf{V}$, e.g. in a summation. We follow the convention of using upper case for variables, lower case for instantiations/values and boldface for sets.

A *factor* f over variables $\mathbf{V}$ is a function that maps every instantiation $\mathbf{v}{\in}\mathbf{V}$ to a number $f(\mathbf{v})$ (often a probability). It is similar to an ordinary mathematical function with multiple arguments, only it refers to them by name instead of position. For example, where for ordinary functions in general $f(x, y) \neq f(y, x)$, for factors it is the case that $f(X{=}x, Y{=}y) = f(Y{=}y, X{=}x)$; formally, the factor f is applied to the set $\{X{=}x, Y{=}y\}$, but we omit the braces to reduce clutter.

In inference implementations, factors are stored as multidimensional arrays. Where for ordinary functions these dimensions would be *numbered* conforming to the function arguments, for factors they are *named* after variables, hence it seems natural to define f's dimensionality as the whole set: $\dim(f) \stackrel{\text{def}}{=} \mathbf{V}$. The *weight* of a factor equals the size of the array needed to store all its values: $\mathsf{weight}(f) \stackrel{\text{def}}{=} \prod_{V_j \in \dim(f)} |\mathrm{dom}(V_j)|$.

Factor algebra provides the tools for manipulating factors:

- *Application* $f(\mathbf{v})$: Applying a factor f to an instantiation $\mathbf{v}$ with $\mathbf{v}{\in}\dim(f)$ is simply function application, and results in a single value. However, a factor can also be partially applied, i.e. to an instantiation $\mathbf{w} \in \mathbf{W} \subset \dim(f)$. The result of this operation is a factor $f' = f(\mathbf{w})$ with $\dim(f') = \dim(f) \setminus \mathbf{W}$. Superfluous variables ($\notin \dim(f)$) are simply ignored: $f(\mathbf{v}, V_{n+1}{=}x) = f(\mathbf{v})$.

$$(f(\mathbf{w}))(\mathbf{u}) \stackrel{\text{def}}{=} f(\mathbf{u},\mathbf{w})$$
$$(f \otimes g)(\mathbf{v}) \stackrel{\text{def}}{=} f(\mathbf{v}) \cdot g(\mathbf{v})$$
$$(\Sigma_{\mathbf{W}} f)(\mathbf{u}) \stackrel{\text{def}}{=} \sum_{\mathbf{w}\in\mathbf{W}} f(\mathbf{u},\mathbf{w})$$

$$f[V{=}d](\mathbf{u}) \stackrel{\text{def}}{=} f(\mathbf{u}, V{=}d(\mathbf{u}))$$
$$\mathbb{1}_{V=d}(V{=}v,\mathbf{u}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } v = d(\mathbf{u}) \\ 0 & \text{if } v \neq d(\mathbf{u}) \end{cases}$$

Fig. 1: Factor algebra. First, the commonly found operations *application*, *multiplication* and *summation*; next, our new operations *indexing* and *concretization*.

$$f \otimes \mathbb{1} = f \tag{1}$$
$$f \otimes g = g \otimes f \tag{2}$$
$$f \otimes (g \otimes h) = (f \otimes g) \otimes h = \bigotimes\{f,g,h\} \tag{3}$$
$$\Sigma_V \Sigma_W f = \Sigma_W \Sigma_V f = \Sigma_{V,W} f \tag{4}$$
$$\Sigma_V (f \otimes g) = \Sigma_V f \otimes g \quad \text{if } V \in \dim(f),\ V \notin \dim(g) \tag{5}$$
$$\Sigma_V (f \otimes g) = f \otimes \Sigma_V g \quad \text{if } V \notin \dim(f),\ V \in \dim(g) \tag{6}$$
$$\big(f \otimes g\big)(\mathbf{e}) = f(\mathbf{e}) \otimes g(\mathbf{e}) \tag{7}$$
$$\big(\Sigma_V f\big)(\mathbf{e}) = \Sigma_V f(\mathbf{e}) \quad \text{if } \mathbf{e} \text{ does not instantiate } V \tag{8}$$

Fig. 2: Laws of factor algebra.

- *Multiplication* $f \otimes g$: Multiplication of factors is lifted value multiplication. The result of this operation is a factor $h = f \otimes g$ with $\dim(h) = \dim(f) \cup \dim(g)$. Multiplication is associative and commutative; we write $\bigotimes_{1\leq j\leq n} f_j = f_1 \otimes f_2 \otimes \ldots \otimes f_n$ for n-way multiplication, and $\mathbb{1}$ for its unit element.
- *Summation* $\Sigma_{\mathbf{W}} f$: Summation over variables $\mathbf{W}$ removes the $\mathbf{W}$ variables from the dimensionality of a factor by summing up all the values of instantiations that differ only at $\mathbf{W}$ variables.

Formal definitions of these operations are listed in Fig. 1 (top). They obey several equality laws (Fig. 2) which we will use to prove correctness of inference.

As our work is expressed in factor algebra, we go as far as to *define* Bayesian networks in terms of factor algebra, in order to keep notation as coherent as possible. A *Bayesian network* is a triple $(\mathbf{V}, \mathsf{par}, \mathbf{cpd})$, where $\mathbf{V}$ consists of n discrete variables as above; the function par maps each variable V_j to a set of *parents* $\mathbf{V}_{\mathsf{par}(j)} \subset \mathbf{V}$ in such a way that there are no cycles; the set $\mathbf{cpd} = \{cpd_1, \ldots, cpd_n\}$ contains, for each V_j, a factor cpd_j over $\{V_j\} \cup \mathbf{V}_{\mathsf{par}(j)}$ with $(\Sigma_{V_j} cpd_j)(\mathbf{v}_{\mathsf{par}(j)}) = 1$ for each $\mathbf{v}_{\mathsf{par}(j)} \in \mathbf{V}_{\mathsf{par}(j)}$. Each Bayesian network defines a factor jpd known as its *joint probability distribution*: $jpd \stackrel{\text{def}}{=} \bigotimes_{1\leq j\leq n} cpd_j$. It can be proven that $\Sigma_{\mathbf{V}} jpd = 1$. An example (fragment) of a Bayesian network with $\mathbf{V} = \{X, Y, Z, A, M, \ldots\}$ is shown in Fig. 3, and applying the $\otimes$ and Σ operations to its $\mathbf{cpd}$ factors is demonstrated in Fig. 4.

An *inference query* is defined as the joint distribution over a set of query variables $\mathbf{Q} \subseteq \mathbf{V}$ and an instantiation $\mathbf{e}$ of evidence variables $\mathbf{E} \subseteq \mathbf{V}$:

$$inf_{\mathbf{Q},\mathbf{e}} \stackrel{\text{def}}{=} (\Sigma_{\mathbf{R}} jpd)\,(\mathbf{e}) \qquad \text{where } \mathbf{R} = \mathbf{V} \setminus (\mathbf{Q} \cup \mathbf{E})$$

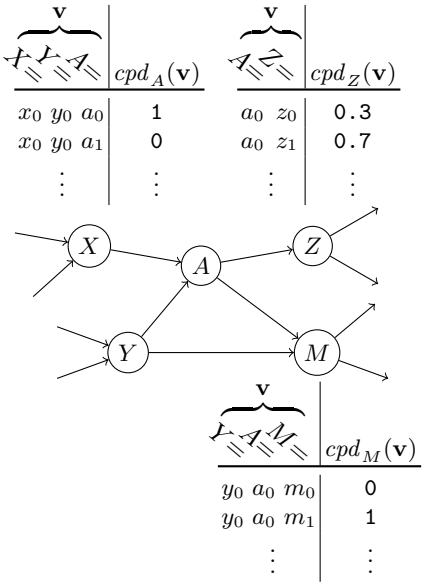

Fig. 3: Example fragment of a Bayesian network, in which the *cpd* factors for variables A, Z and M are partially given. An array implementation of the factors would store the values in `typewriter` font.

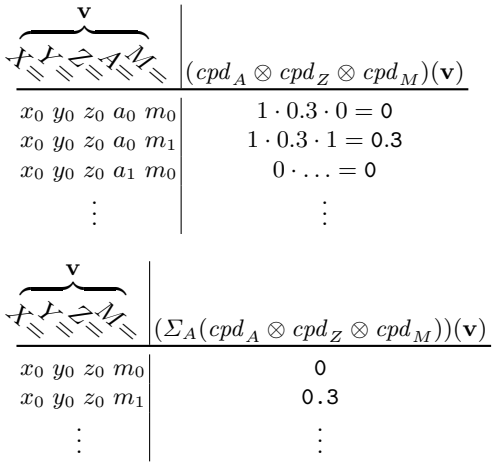

Fig. 4: Examples of applying the factor algebra operators $\otimes$ and Σ_A to the factors from the Bayesian network. Note: as $X{=}x_0$ and $Y{=}y_0$ together determine that $A{=}a_0$, the first two values in the bottom factor equal those in the top factor.

An *inference procedure* is an algorithm that, given an arbitrary Bayesian network, query variables $\mathbf{Q}$ and evidence $\mathbf{e}$, calculates the value of $\mathit{inf}_{\mathbf{Q},\mathbf{e}}$.

Relation to the conventional definition. The *jpd* factor corresponds to the joint probability distribution in the conventional definition: $\mathsf{P}(\mathbf{v}) = \mathit{jpd}(\mathbf{v})$, and it is easily shown that the conditional probability distributions $\mathsf{P}(v_j|\mathbf{v}_{\mathsf{par}(j)})$ derived from this joint distribution equal the cpd_j factors. The reason that we avoid the P notation is that it often clashes with factor application. For example, for $\mathbf{W} \subset \mathbf{V}$, $\mathsf{P}(\mathbf{W})$ denotes a distribution over $\mathbf{W}$ while $\mathit{jpd}(\mathbf{w})$ is a factor over the complement $\mathbf{V} \setminus \mathbf{W}$. The two notations are related as follows: for a marginal distribution, $\mathsf{P}(\mathbf{W}) = \Sigma_{\mathbf{V}\setminus\mathbf{W}}\mathit{jpd}$, and for partial evidence, $\mathsf{P}(\mathbf{e}) = (\Sigma_{\mathbf{V}\setminus\mathbf{E}}\mathit{jpd})(\mathbf{e})$.

3 Variable Elimination as Factor Rewriting

In this section, we review the inference procedure of variable elimination[11]. More precisely, variable elimination is a family of inference procedures parameterized by a *variable elimination order* which mostly determines its efficiency. The order we use is the *minweight* heuristic, known in practice to outperform other heuristics when variables have different domain sizes[7].

Algorithm 1: Minweight variable elimination.

Input: – Bayesian network $(\mathbf{V}, \mathsf{par}, \{cpd_1, \ldots, cpd_n\})$
– query variables $\mathbf{Q} \subseteq \mathbf{V}$; evidence $\mathbf{e}$ (instantiation of $\mathbf{E} \subseteq \mathbf{V}$)
Output: *result*, a factor over $\mathbf{Q}$ equal to $inf_{\mathbf{Q},\mathbf{e}}$

$\mathbf{W} := \mathbf{V} \setminus (\mathbf{Q} \cup \mathbf{E})$
foreach cpd_j **do** $f_j := cpd_j(\mathbf{e})$

while $\mathbf{W}$ is not empty **do**
  choose $V_i \in \mathbf{W}$ for which the cost of $eliminate(V_i)$ is smallest
  $eliminate(V_i)$
  $\mathbf{W} := \mathbf{W} \setminus \{V_i\}$

$result := \bigotimes\{\text{all remaining } f_j\}$

procedure $eliminate(V_i)$
  $p := \mathbb{1}$
  foreach f_j s.t. $V_i \in \dim(f_j)$ **do**
    $p := p \otimes f_j$
    delete f_j
  $f_i := \Sigma_{V_i} p$

The algorithm, its correctness proof and the rationale for the heuristic presented here serve as a basis for our extended version, which will be introduced in Sec. 4 and exploits deterministic variables.

Variable elimination (Alg. 1) roughly proceeds as follows: it starts out from the collection of cpd_j factors and directly applies the evidence $\mathbf{e}$; then it repeatedly selects a variable V_i from $\mathbf{R}$, removes the factors containing V_i from the collection and replaces them by their product, with V_i summed out. These steps are repeated until all variables from $\mathbf{R}$ have been eliminated. Note that at any time, the remaining factors are stored in the f_j variables (where the j indices are in general not contiguous, as these variables are progressively deleted); this scheme is chosen for the sake of extension into Alg. 2.

Postponing the question of the variable elimination order (i.e. selecting the smallest cost) for a while, we first set out to prove that Alg. 1 produces the correct result: a factor algebra expression that is equal to $inf_{\mathbf{Q},\mathbf{e}}$.

Proof. Specifically, the invariant $inf_{\mathbf{Q},\mathbf{e}} = \Sigma_{\mathbf{W}} \bigotimes\{\text{all remaining } f_j\}$ holds at the start of each loop iteration. At the first iteration,

$$\Sigma_{\mathbf{W}} \bigotimes\{\text{all remaining } f_j\} = \Sigma_{\mathbf{R}} \bigotimes_{1 \leq i \leq n} cpd_i(\mathbf{e}) = \left(\Sigma_{\mathbf{R}} \bigotimes_{1 \leq i \leq n} cpd_i\right)(\mathbf{e})$$

with the last equality due to laws (7) and (8). This equals $inf_{\mathbf{Q},\mathbf{e}}$ by definition. Next, by law (6):

$$\Sigma_{\mathbf{W}} \left(\bigotimes_{V_i \notin \dim(f_j)} f_j \otimes \bigotimes_{V_i \in \dim(f_j)} f_j \right) = \Sigma_{\mathbf{W} \setminus V_i} \left(\bigotimes_{V_i \notin \dim(f_j)} f_j \otimes \Sigma_{V_i} \bigotimes_{V_i \in \dim(f_j)} f_j \right)$$

Assume that the invariant holds at the start of the loop, so $inf_{\mathbf{Q},\mathbf{e}}$ equals the left expression (in which we have divided up 'all remaining f_j' for convenience into those that do and do not contain V_i). Now, at the end of the loop, $\mathbf{W}$ will be set to $\mathbf{W} \setminus \{V_i\}$, and the f_j factors with $V_i \in \dim(f_j)$ will be replaced with $f_i := \Sigma_{V_i} \bigotimes_{V_i \in \dim(f_j)} f_j$. Thus, the expression at the right is equal to $\Sigma_{\mathbf{W}} \bigotimes \{\text{all remaining } f_j\}$ for the values of $\mathbf{W}$ and f_j at the start of the next loop, and the invariant holds again. After the last loop, $\mathbf{W}$ is empty, so $inf_{\mathbf{Q},\mathbf{e}} = \bigotimes \{\text{all remaining } f_j\} = \mathit{result}$. □

Having established that the algorithm produces a correct result, let us examine what this result actually consists of. Although Alg. 1 can certainly be read to perform array operations at factor assignments such as $p := p \otimes f_j$ and $f_i := \Sigma_{V_i} p$, and return an array with the correct values at the end, it does not have to perform any array operations *at all*. Instead, we prefer the reading where it performs a *symbolic construction* of a new factor algebra expression at these points. In that case, the result of the algorithm is not an array, but a large symbolic expression which, as we have just proven, is a rewriting of $inf_{\mathbf{Q},\mathbf{e}}$. It can be evaluated at a later stage to produce said array. Thus, the inference procedure is divided into a *rewrite* phase and an *evaluation* phase.

The purpose of rewriting $inf_{\mathbf{Q},\mathbf{e}}$ is that the resulting expression is somehow 'more efficient', i.e. needs less time or space to execute. For this statement to make any sense, we need to ascribe an *operational semantics* to factor algebra expressions (as opposed to the denotational semantics of Fig. 1, in which both expressions are equivalent, as they have the same value). This is easily done (e.g. to perform $f \otimes g$, first perform f and store its resulting array somewhere, then do the same for g, and finally construct the result array by multiplying the stored values); we do not elaborate any further on this. With an operational semantics in place, one can define a cost function in terms of space or time needed.

The order in which Alg. 1 picks variables can now be explained in terms of this cost function: at each iteration, it greedily picks the one which is cheapest to eliminate (paying no attention to the effect that this might have on eliminating other variables later on). To model the cost of an elimination step, we use the size of the largest array constructed in that step, i.e. $\mathsf{weight}(p)$ for the final value of p in $\mathit{eliminate}(V_i)$. We define the cost like this in order to simulate the existing minweight heuristic; it might pay off to consider other cost functions. Note that, as $\mathsf{weight}(p)$ can be determined from the symbolic expression p, the cost of $\mathit{eliminate}(V_i)$ can be calculated without actually executing p.

This evokes an interesting parallel to query optimization in database management systems, in which it is common practice to construct and optimize a query plan before executing it. Drawing on this parallel, it has already been shown that variable elimination plans can be further optimized using database techniques[2].

4 Factor Indexing

This section presents the main contribution of this article: *factor indexing*, and its integration in variable elimination. We propose a rewrite rule for marginalization without summation, and express it in factor algebra by introducing a new indexing operation. Then, we extend Alg. 1 into Alg. 2, which uses this rule to eliminate deterministic variables.

A variable $Y \in \mathbf{V}$ is called *deterministic* if its value is functionally determined by the value of its parents (here $\mathbf{X} \subset \mathbf{V}$). This means that its conditional probability distribution has the following form:

$$cpd_Y(Y{=}y, \mathbf{x}) = \begin{cases} 1 & \text{if } y = d_Y(\mathbf{x}) \\ 0 & \text{if } y \neq d_Y(\mathbf{x}) \end{cases}$$

where d_Y is a factor over $\mathbf{X}$ with values in $\mathrm{dom}(Y)$, which we call Y's *deterministic factor*. To make deterministic variables explicit, we now extend the definition of a Bayesian network to $(\mathbf{V}, \mathsf{par}, \mathbf{cpd}, \mathbf{d})$: the set $\mathbf{cpd}$ still contains the factors for non-deterministic variables $V_1, \ldots, V_m$, and the new set $\mathbf{d}$ contains the deterministic factors for deterministic variables $V_{m+1}, \ldots, V_n$. So, factor d_j must be a factor over $\mathsf{par}(V_j)$ with values in $\mathrm{dom}(V_j)$ (unlike a cpd_j factor, $V_j \notin \dim(d_j)$). An example is shown in Fig. 5; it is the same Bayesian network fragment as in Fig. 3, but now with explicit deterministic variables.

To emphasize the simplicity and generality of our rewrite rule, we first present it in conventional notation for our running example. We show the *marginalization without summation* of deterministic variable A:

$$\sum_{a \in A} \mathsf{P}(a|x,y)\mathsf{P}(z|a)\mathsf{P}(m|a,y) \;=\; \mathsf{P}(z|A{=}d_A(x,y))\mathsf{P}(m|A{=}d_A(x,y),y)$$

It relies on the following observation: given a certain x and y, the summation contains only one nonzero term, because there is only one value for a that makes $\mathsf{P}(a|x,y)$ nonzero. So, instead of summing, we can just substitute this value $d_A(x,y)$ for every a.

Note that the left-hand side of the formula corresponds to the calculations made by Alg. 1 when eliminating A: first the product $cpd_A \otimes cpd_Z \otimes cpd_M$ is created (a factor over 5 variables), which is subsequently reduced by summing. The deterministic variable elimination that we will define corresponds to the right-hand side; to eliminate A, all we need to do is construct the two factors over variables XYZ and MXY. Their multiplication can be postponed.

To use the above substitution in factor algebra, we introduce new operations:

- *Indexing* $f[V{=}d]$: Factor f is indexed in dimension $V \in \dim(f)$ by another factor d, which has values in $\mathrm{dom}(V)$. The resulting factor $f' = f[V{=}d]$ has $\dim(f') = (\dim(f) \setminus \{V\}) \cup \dim(d)$. Contrary to conventional indexing, the dimensionality of f' can be larger than that of f.
- *Concretization* $\mathbb{1}_{V=d}$: Deterministic factor d is turned into a 'probabilistic' factor over the variables $\{V\} \cup \dim(d)$, whose value is 1 wherever the value of variable V matches d's function value.

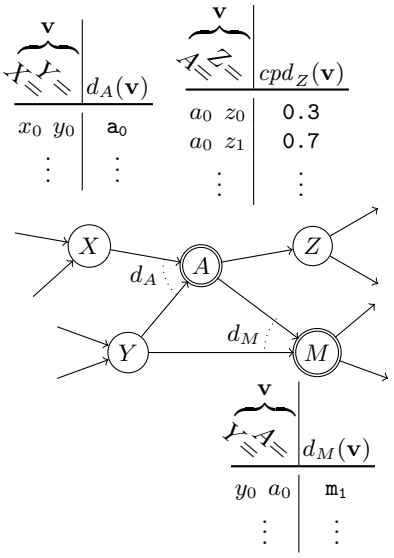

Fig. 5: Same fragment as Fig. 3, with explicit deterministic variables A,M. Their *deterministic factors* store states of A,M instead of probabilities.

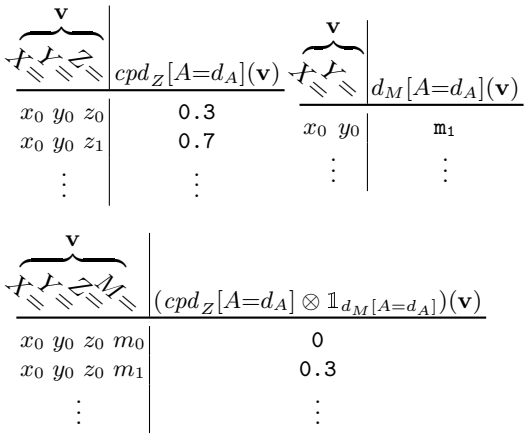

Fig. 6: Applying the factor indexing operation to the cpd_Z and d_M factors from the Bayesian network. Their product equals the summed factor in Fig. 4.

Formal definitions can be found in Fig. 1 (bottom), and examples in Fig. 6. Note that the definition of concretization corresponds to the factor containing the conditional probability distribution of a deterministic variable. So, translating a network with explicit deterministic variables into a conventional one is done by setting $cpd_j = \mathbb{1}_{V_j=d_j}$ for all $m < j \leq n$. We include the concretization operation because, in general, not every deterministic variable is eliminated by factor indexing in our algorithm: sometimes it is necessary to treat its deterministic factor as a conventional one.

With these definitions in place, we can express the above (example) rewrite rule in factor algebra:

$$\Sigma_A(\mathbb{1}_{A=d_A} \otimes \mathbb{1}_{M=d_M} \otimes cpd_Z) \;=\; \mathbb{1}_{M=d_M[A=d_A]} \otimes cpd_Z[A{=}d_A]$$

This formulation merits some clarification. As we cannot directly multiply deterministic and probabilistic factors with each other, the equality is stated at a 'probabilistic' level, i.e. with all deterministic factors concretized. An additional complication is that A is not the only deterministic variable involved: M is deterministic as well. We can choose to ignore this fact, i.e. treat M as probabilistic and just index its concretization, which results in the factor $\mathbb{1}_{M=d_M}[A{=}d_A]$. However, this turns out to equal $\mathbb{1}_{M=d_M[A=d_A]}$, in which the deterministic factor d_M is indexed by another deterministic factor d_A. Unlike with multiplication,

this is no problem; see also Fig. 6. Consequently, we do not have to concretize the d_M factor when eliminating A in our elimination algorithm.

As we mentioned, the above rewrite rule can be generalized to any deterministic variable. The elimination of deterministic variable V_i from a product of factors f_j and deterministic factors d_j can be rewritten without Σ_{V_i} summation:

$$\Sigma_{V_i}\left(\mathbb{1}_{V_i=d_i} \otimes \bigotimes_{V_j \in \dim(f_j)} f_j \otimes \bigotimes_{V_j \in \dim(d_j)} \mathbb{1}_{V_j=d_j}\right) = \bigotimes_{V_j \in \dim(f_j)} f_j[V_i{=}d_i] \otimes \bigotimes_{V_j \in \dim(d_j)} \mathbb{1}_{V_j=d_j[V_i=d_i]}$$

This rewrite rule can be seen as an addition to the laws in Fig. 2. It provides more possibilities for rewriting an inference query, and can as such be used in any inference procedure that makes use of factor operations. Here, we apply it in a variable elimination algorithm with factor indexing (Alg. 2). It has the same structure as Alg. 1, but is extended as follows:

- For a deterministic variable V_i, we store d_i instead of $\mathbb{1}_{V_i=d_i}$.
- To eliminate a deterministic variable V_i, we use the rewrite rule: we index all currently existing f_j and d_j factors over V_i by $V_i{=}d_i$, and delete d_i itself.
- Not all deterministic variables are eliminated like this: during the elimination of a non-deterministic variable V_i, all deterministic factors over V_i have to be concretized. Also, for a deterministic *evidence* variable, its factor is concretized during initialization.

The proposed elimination heuristic is again the *cost* of the next elimination step. However, we update the definition of this cost to reflect the size of the factors produced in this step. If V_i has *no* deterministic factor d_j associated with it, the cost is still $\mathsf{weight}(p)$. If it *does*, we define the cost to be

$$\sum_{V_i \in \dim(f_j)} \mathsf{weight}(f_j[V_i{=}d_i]) + \sum_{V_i \in \dim(d_j)} \mathsf{weight}(d_j[V_i{=}d_i])$$

For Alg. 2, a correctness proof is similar to the one above can be given. Its invariant is:

$$\mathit{inf}_{\mathbf{Q},\mathbf{e}} = \Sigma_{\mathbf{W}}\left(\left(\bigotimes\{\text{all remaining } f_j\}\right) \otimes \bigotimes\left\{\ \mathbb{1}_{V_j=d_j} \,\middle|\, \text{all remaining } d_j\ \right\}\right)$$

5 Empirical Evaluation

We have implemented the factor algebra described above in Python, using the package NumPy which provides an n-dimensional array and executes array operations using fast C loops (not unlike MATLAB). The $\otimes$ operator directly translates to NumPy's array multiplication, which can handle the situation where the operands have different dimensions. Indexing an array with another array is supported in NumPy as well.

Algorithm 2: Minweight variable elimination with factor indexing.

Input: – Bayesian network w/ det. vars $(\mathbf{V}, \mathsf{par}, \{cpd_1, \ldots, cpd_m\}, \{d_{m+1}, \ldots, d_n\})$
– query variables $\mathbf{Q} \subseteq \mathbf{V}$; evidence $\mathbf{e}$ (instantiation of $\mathbf{E} \subseteq \mathbf{V}$)
Output: *result*, a factor over $\mathbf{Q}$ equal to $inf_{\mathbf{Q},\mathbf{e}}$

$\mathbf{W} := \mathbf{V} \setminus (\mathbf{Q} \cup \mathbf{E})$
foreach cpd_j **do** $f_j := cpd_j(\mathbf{e})$
foreach d_j **do**
  if $V_j \in \mathbf{E}$ **then**
    $f_j := \mathbb{1}_{V_j = d_j}(\mathbf{e})$
  else
    $d_j := d_j(\mathbf{e})$

while $\mathbf{W}$ is not empty **do**
  choose $V_i \in \mathbf{W}$ for which the cost of *eliminate*(V_i) is smallest
  eliminate(V_i)
  $\mathbf{W} := \mathbf{W} \setminus \{V_i\}$

$result := (\bigotimes\{\text{all remaining } f_j\}) \otimes \bigotimes \{\ \mathbb{1}_{V_j = d_j} \mid \text{all remaining } d_j\ \}$
procedure *eliminate*(V_i)
  if d_i exists **then**
    foreach d_j s.t. $V_i \in \dim(d_j)$ **do** $d_j := d_j[V_i = d_i]$
    foreach f_j s.t. $V_i \in \dim(f_j)$ **do** $f_j := f_j[V_i = d_i]$
    delete d_i
  else
    $p := \mathbb{1}$
    foreach d_j s.t. $V_i \in \dim(d_j)$ **do**
      $p := p \otimes \mathbb{1}_{V_j = d_j}$
      delete d_j
    foreach f_j s.t. $V_i \in \dim(f_j)$ **do**
      $p := p \otimes f_j$
      delete f_j
    $f_i := \Sigma_{V_i} p$

We perform inference on 4 networks with deterministic nodes known from the Bayesian network literature (the students network is from the UAI'08 evaluation track). We also investigated 6 generated networks of 100 nodes, with 30 root nodes and 70 nodes with 2 parents (randomly chosen from earlier generated nodes). Each node has randomly generated probabilities; each of the 70 non-root nodes has a chance of being deterministic, in which case we randomly generate a deterministic function. Each variable has the same domain; between networks, we vary the domain size (2 or 4). Also, we vary the fraction of deterministic nodes (30%, 60%, 90% of the non-root nodes).

For each network, we take medians over 10 runs; in each run, we instantiate 10 randomly chosen[1] evidence variables $\mathbf{e}$ and choose one random query variable Q. Then we use algorithms Alg. 1 and Alg. 2 to generate a symbolic expression (a

[1] However, for students, we took the 9 easiest evidence files from the UAI'08 evaluation.

Table 1: Experimental results. Numbers are median values over 10 random queries.

network	# vars (det.)	plan cost		cost impr.	eval. time (s)		speedup
		Alg. 1	Alg. 2	Alg. 1/Alg. 2	Alg. 1	Alg. 2	Alg. 1/Alg. 2
munin-1	189 (65)	278M	260M	1.00	6.94	7.91	0.935
munin-4	1041 (411)	23.3M	19.2M	1.22	0.481	0.382	1.25
diabetes	413 (24)	13.2M	13.1M	1.00	0.148	0.151	0.994
students	376 (304)	4.32M	14.7K	293	0.205	0.053	4.13
random-2-30	100 (±21)	16.3K	3.85K	2.91	0.0120	0.0106	1.15
random-2-60	100 (±42)	19.6K	2.47K	5.82	0.0121	0.0088	1.35
random-2-90	100 (±63)	14.6K	0.711K	15.0	0.0117	0.0064	1.90
random-4-30	100 (±21)	6.28M	2.38M	9.23	0.122	0.0536	5.38
random-4-60	100 (±42)	2.27M	49.0K	55.1	0.0504	0.0098	5.39
random-4-90	100 (±63)	4.41M	14.7K	257	0.0908	0.0065	16.3

plan) for $inf_{\mathbf{Q},\mathbf{e}}$, i.e. we execute them as a *rewrite phase* as discussed in Sect. 3. As it is completely implemented in Python (without regard for speed), we do not time this phase; its performance would severely distort the overall timing results.

We record the cost of the generated plans, i.e. the summed weight of all the intermediate factors. In the second phase, we evaluate the plans and record the (wall clock) duration. The experiments were performed on a machine with a 3GHz Intel Core2Duo processor and 2GB RAM.

Results are shown in Table 1: the factor indexing technique provides speedups ranging up to 16×. Expectations are confirmed that it works best with a high fraction of deterministic nodes and/or larger domain sizes. However, we noticed that the variance in performance between runs can be high: we suspect that the current heuristic can easily guide the algorithm in the wrong way, and will investigate more robust heuristics in the future.

6 Conclusions and Future Work

We propose a new variable elimination technique for exact inference on Bayesian networks, in which deterministic variables are eliminated not by summation but by a factor indexing operation. We emphasize the role of factor algebra, which enables (a) a concise definition of the algorithm, (b) a straightforward correctness proof, and (c) a model for defining an elimination order heuristic in terms of the *cost* of array operations. Indeed, our updated minweight heuristic has little to do with the network's graph structure anymore; this is in line with common knowledge that treewidth is not so important for highly deterministic networks.

A preliminary empirical evaluation shows that the technique performs decently on real-world networks (small speedups) and good on randomly generated networks (speedups of 1–16). We expect much room for improvement here: first, by developing heuristics that take into account the *actual* cost of performing the different array operations instead of the size of the resulting array; second, by

exploiting low-level machine knowledge to decrease these actual costs (building on the connection between inference optimization and database research which we have pointed out). For example, current CPUs and GPUs often feature vectorized processing modes, which we expect can be exploited for the bulk array operations of probabilistic inference. When used properly, this might outperform inference techniques for determinism that cannot be expressed as array operations, e.g. [1, 8].

Furthermore, we argue that our technique has much potential for combination with other inference algorithms that use factor operations, e.g. junction tree propagation[9], recursive conditioning[3] and factor decomposition techniques[5, 6, 10].

Acknowledgements

The authors have been supported by the OCTOPUS project under the responsibility of the Embedded Systems Institute. The OCTOPUS project is partially supported by the Netherlands Ministery of Economic Affairs under the Embedded Systems Institute program.

References

1. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. 172(6-7), 772–799 (2008)
2. Corrada Bravo, H., Ramakrishnan, R.: Optimizing MPF queries: decision support and probabilistic inference. In: SIGMOD Conference. pp. 701–712 (2007)
3. Darwiche, A.: Recursive conditioning. Artif. Intell. 126(1-2), 5–41 (2001)
4. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artif. Intell. 113(1-2), 41–85 (1999)
5. Díez, F.J., Galán, S.F.: Efficient computation for the Noisy MAX. Int. J. Intell. Syst. 18(2), 165–177 (2003)
6. Heckerman, D., Breese, J.S.: Causal independence for probability assessment and inference using Bayesian networks. IEEE Transactions on Systems, Man and Cybernetics, Part A 26(6), 826–831 (1996)
7. Kjærulff, U.: Triangulation of graphs — algorithms giving small total state space. Tech. Rep. R-90-09, Dept. of Mathematics and Computer Science, Aalborg University (1990), http://www.cs.aau.dk/ uk/papers/R90-09.ps.gz
8. Larkin, D., Dechter, R.: Bayesian inference in the presence of determinism. In: Bishop, C.M., Frey, B.J. (eds.) Proceedings of Ninth International Workshop on Artificial Intelligence and Statistics. Key West, USA (2003)
9. Lauritzen, S.L., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems. Journal of the Royal Statistical Society. Series B 50(2), 157–224 (1988)
10. Vomlel, J.: Exploiting functional dependence in Bayesian network inference. In: Darwiche, A., Friedman, N. (eds.) UAI '02, Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence, University of Alberta, Edmonton, Alberta, Canada. pp. 528–535. Morgan Kaufmann (2002)
11. Zhang, N.L., Poole, D.: Exploiting causal independence in bayesian network inference. J. Artif. Intell. Res. (JAIR) 5, 301–328 (1996)