ECAI 2020
G.D. Giacomo et al. (Eds.)

© 2020 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA200391

An Advance on Variable Elimination with Applications to Tensor-Based Computation

Adnan Darwiche[1]

Abstract. We present new results on the classical algorithm of variable elimination, which underlies many algorithms including for probabilistic inference. The results relate to exploiting functional dependencies, allowing one to perform inference efficiently on models that have very large treewidth. The highlight of the advance is that it works with standard (dense) factors, without the need for sparse factors or techniques based on knowledge compilation that are commonly utilized. This is significant as it permits a direct implementation of the improved variable elimination algorithm using tensors and their operations, leading to extremely efficient implementations especially when learning model parameters. We illustrate the efficacy of our proposed algorithm by compiling Bayesian network queries into tensor graphs and then learning their parameters from labeled data using a standard tool for tensor computation.

1 Introduction

The work reported in this paper is motivated by an interest in model-based supervised learning, in contrast to model-free supervised learning that currently underlies most applications of neural networks. We briefly discuss this subject first to put the proposed work in context.

Supervised learning has become very influential recently and stands behind most real-world applications of AI. In supervised learning, one learns a *function* from labeled data, a practice that is now dominated by the use of neural networks to represent such functions; see [16, 17, 1, 26]. Supervised learning can be applied in other contexts as well, such as causal models in the form of Bayesian networks [23, 24, 25]. In particular, for each query on the causal model, one can *compile* an Arithmetic Circuit (AC) that maps evidence (inputs) to the posterior probability of interest (output) [9, 10]. AC parameters, which correspond to Bayesian network parameters, can then be learned from labeled data using gradient descent. Hence, like a neural network, the AC is a circuit that computes a function whose parameters can be learned from labeled data.

The use of ACs in this fashion can be viewed as *model-based* supervised learning, in contrast to *model-free* supervised learning using neural networks. Model-based supervised learning is attractive since the AC can integrate the *background knowledge* embecded in its underlying causal model. This has a number of advantages, which include a reduced reliance on data, improved robustness and the ability to provide data-independent guarantees on the learned function. One important type of background knowledge is functional dependencies between variables and their direct causes in a model (a special case of what is known as *determinism*). Not only can this type of knowledge significantly reduce the reliance on data, but it can also significantly improve the complexity of inference. In fact, substantial efforts have been dedicated to exploiting determinism in probabilistic inference, particularly the compilation of Bayesian networks into ACs [9, 3, 2], which is necessary for efficient inference on dense models.

There are two main approaches for exploiting functional dependencies. The first is based on the classical algorithm of variable elimination (VE), which underlies algorithms for probabilistic inference including the *jointree* algorithm [29, 11, 18]. VE represents a model using *factors,* which are tables or multi-dimensional arrays. It then performs inference using a few and simple *factor operations.* Exploiting functional dependencies within VE requires *sparse factors*; see, e.g., [19, 21]. The second approach for exploiting functional dependencies reduces probabilistic inference to *weighted model counting* on a propositional formula that encodes the model, including its functional dependencies. It then *compiles* the formula into a circuit that is tractable for model counting; see, e.g., [8, 2]. This approach is in common use today given the efficacy of knowledge compilers.

Our main contribution is a new approach for exploiting functional dependencies in VE that works with *standard (dense) factors.* This is significant for the following reason. We wish to map probabilistic inference, particularly the learning of parameters, into a *tensor computation* to exploit the vast progress on tensor-based technology and be on par with approaches that aggressively exploit this technology. Tensors are multi-dimensional arrays whose operations are heavily optimized and can be extremely fast, even on CPU-based platforms like modern laptops (let alone GPUs). A tensor computation takes the form of a *tensor graph* with nodes representing tensor operations. Factors map directly to tensors and sparse factors to sparse tensors. However, sparse tensors have limited support in state of the art tools, which prohibits an implementation of VE using sparse tensors.[2] Knowledge compilation approaches produce circuits that cast into *scalar tensor graphs,* which are less effective than general tensor graphs as they are less amenable to parallelization. Moreover, while our approach needs to know that there is a functional dependency between variables it does not require the specific dependency (the specific numbers). Hence, it can be used to speed up inference even when the model parameters are unknown which can be critical when learning model parameters from data. Neither of the previous approaches can exploit this kind of abstract information.

VE is based on two theorems that license factor operations. We add two new theorems that license more operations in the presence of functional dependences. This leads to a standard VE algorithm except with a significantly improved complexity and computation that maps directly to a tensor graph. We present experimental results for

[1] University of California, Los Angeles, email: darwiche@cs.ucla.edu

[2] For example, in TensorFlow, a sparse tensor can only be multiplied by a dense tensor, which rules out the operation of (sparse) factor multiplication that is essential for sparse VE; see `https://www.tensorflow.org/api_docs/python/tf/sparse/SparseTensor`.

inference and learning that show promise of the proposed algorithm.

We start in Section 2 by discussing factors, their operations and the VE algorithm including its underlying theorems. We also present our new VE theorems in this section. We then propose a new VE algorithm in Section 3 that exploits functional dependencies. We show how the proposed algorithm maps to tensor graphs and why this matter in Section 4. We follow by case studies in Section 5 that illustrate the algorithm's performance in the context of model-based supervised learning. We finally close with some remarks in Section 6.

2 The Fundamentals: Factors & Operations

The VE algorithm is based on applying operations to factors.

A *factor* for discrete variables $\mathbf{X}$ is a function that maps each instantiation $\mathbf{x}$ of variables $\mathbf{X}$ into a number. The following are two factors over binary variables A, B, C and ternary variable D:

A	D	$f(AD)$
0	0	0.2
0	1	0.3
0	2	0.6
1	0	0.9
1	1	0.6
1	2	0.1

A	B	C	$g(ABC)$
0	0	0	1.0
0	0	1	0.0
0	1	0	0.0
0	1	1	1.0
1	0	0	0.2
1	0	1	0.8
1	1	0	0.5
1	1	1	0.5

Factors can be represented as multi-dimensional arrays and are now commonly referred to as *tensors* (factor variables corresponds to array/tensor dimensions). One needs three factor operations to implement the VE algorithm: multiplication, sum-out and normalization.

The *product* of factors $f(\mathbf{X})$ and $g(\mathbf{Y})$ is another factor $h(\mathbf{Z})$, where $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$ and $h(\mathbf{z}) = f(\mathbf{x})g(\mathbf{y})$ for the unique instantiations $\mathbf{x}$ and $\mathbf{y}$ that are compatible with instantiation $\mathbf{z}$. *Summing-out* variables $\mathbf{Y} \subseteq \mathbf{X}$ from factor $f(\mathbf{X})$ yields another factor $g(\mathbf{Z})$, where $\mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$ and $g(\mathbf{z}) = \sum_{\mathbf{y}} f(\mathbf{yz})$. We use $\sum_{\mathbf{Y}} f$ to denote the resulting factor g. We also use $\sum^{=}_{\mathbf{Z}} f$ which reads: sum out all variables from factor f except for variables $\mathbf{Z}$. *Normalizing* factor $f(\mathbf{X})$ yields another factor $g(\mathbf{X})$ where $g(\mathbf{x}) = f(\mathbf{x})/\sum_{\mathbf{x}} f(\mathbf{x})$. We use ηf to denote the normalization of factor f.

A *Bayesian Network (BN)* is specified by a directed acyclic graph (DAG) and a set of factors. In particular, for each node X and its parents $\mathbf{U}$, we need a factor f_X over variables $X\mathbf{U}$. The value $f_X(x\mathbf{u})$ represents the conditional probability $P(x|\mathbf{u})$ and the factor f_X is called a *Conditional Probability Table (CPT)*. The *joint distribution* specified by a Bayesian network is simply the product of its CPTs.

The Bayesian network in Figure 2 has five CPTs $f_A(A)$, $f_B(AB)$, $f_C(AC)$, $f_D(BCD)$ and $f_E(CE)$. The network joint distribution is the product of these factors $Pr(ABCDE) = f_A f_B f_C f_D f_E$.

Evidence on variable X is captured by a factor $\lambda_X(X)$ called an *evidence indicator. Hard evidence* fixes a value x giving $\lambda_X(x) = 1$ and $\lambda_X(x^\star) = 0$ for $x^\star \neq x$. For *soft evidence,* $\lambda_X(x)$ is the *likelihood* of x [23]. The *posterior distribution* of a Bayesian network is the normalized product of its CPTs and evidence indicators.

An expression constructed by applying operations to factors will be called an *f-expression.* Suppose we have evidence on variables A and E in Figure 2. The posterior on variable D is obtained by evaluating the following f-expression:

$$P^\star(D) = \eta \sum_{ABCE} \lambda_A \lambda_E f_A f_B f_C f_D f_E.$$

The VE algorithm factors f-expressions so they are evaluated more efficiently [29, 11] and is based on two theorems; see, e.g., [10, Chapter 6]. The first theorem says that the order in which variables are summed out does not matter.

Theorem 1. $\sum_{\mathbf{XY}} f = \sum_{\mathbf{X}} \sum_{\mathbf{Y}} f = \sum_{\mathbf{Y}} \sum_{\mathbf{X}} f$.

The second theorem allows us to reduce the size of factors involved in a multiplication operation.

Theorem 2. *If variables $\mathbf{X}$ appear in factor f but not in factor g, then $\sum_{\mathbf{X}} f \cdot g = g \sum_{\mathbf{X}} f$.*

Factor $\sum_{\mathbf{X}} f$ is exponentially smaller than factor f so Theorem 2 allows us to evaluate the f-expression $\sum_{\mathbf{X}} f \cdot g$ much more efficiently.

Consider the f-expression $\sum_{ABDE} f(ACE)f(BCD)$. A direct evaluation multiplies the two factors to yield $f(ABCDE)$ then sums out variables $ABDE$. Using Theorem 1, we can arrange the expression into $\sum_{AE} \sum_{BD} f(ACE)f(BCD)$ and using Theorem 2 into $\sum_{AE} f(ACE) \sum_{BD} f(BCD)$, which is more efficient to evaluate.

Using an appropriate order for summing out (eliminating) variables, Theorems 1 and 2 allow one to compute the posterior on any variable in a Bayesian network in $O(n \exp(w))$ time and space. Here, n is the number of network variables and w is the network treewidth (a graph-theoretic measure of the network connectivity).

This works well for sparse networks that have a small treewidth, but is problematic for dense networks like the ones we will look at in Section 5. We present next two new results that allow us to sometimes significantly improve this computational complexity, by exploiting functional relationships between variables and their direct causes.

While we will focus on exploiting functional dependencies in Bayesian networks, our results are more broadly applicable since the VE algorithm can be utilized in many other domains including symbolic reasoning and constraint processing [12]. VE can also be used to *contract* tensor networks which have been receiving increased attention. A *tensor network* is a set of factors in which a variable appears in at most two factors. Contracting a tensor network is the problem of summing out all variables that appear in two factors; see, e.g., [14, 15]. The VE algorithm can also be used to evaluate Einstein summations which are in common use today and implemented in many tools including NumPy.[3]

2.1 Functional CPTs

Consider variable X that has parents $\mathbf{U}$ in a Bayesian network and let factor $f_X(X\mathbf{U})$ be its conditional probability table (CPT).[4] If $f(x\mathbf{u}) \in \{0, 1\}$ for all instantiations x and $\mathbf{u}$, the CPT is said to be *functional* as it specifies a function that maps parent instantiation $\mathbf{u}$ into the unique value x satisfying $f_X(x\mathbf{u}) = 1$. The following CPTs are functional:

X	Y	$f_Y(XY)$
x_0	y_0	0
x_0	y_1	1
x_1	y_0	1
x_1	y_1	0

A	B	$f_B(AB)$
a_0	b_0	0
a_0	b_1	1
a_0	b_2	0
a_1	b_0	0
a_1	b_1	0
a_1	b_2	1

The first specifies the function $x_0 \mapsto y_1$, $x_1 \mapsto y_0$. The second specifies the function $a_0 \mapsto b_1$, $a_1 \mapsto b_2$. Functional dependencies encode a common type of background knowledge (examples in Section 5). They are a special type of *determinism,* which generally refers to the presence of zero parameters in a CPT. A CPT that has zero parameters is not necessarily a functional CPT.

We will next present two results that empower the VE algorithm in the presence of functional CPTs. The results allow us to factor f-expressions beyond what is permitted by Theorems 1 and 2, leading

[3] https://numpy.org/

[4] Since $\sum_x P(x|\mathbf{u}) = 1$ the CPT satisfies $\sum_x f_X(x\mathbf{u}) = 1$ for every $\mathbf{u}$.

to significant reduction in complexity. *The results do not depend on the identity of a functional CPT, only that it is functional.* This is significant when learning model parameters from data.

To state these results, we will use $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$ to denote sets of factors. Depending on the context, a set of factors $\mathcal{F}$ may be treated as one factor obtained by multiplying members of the set $\prod_{f\in\mathcal{F}} f$.

The first result says the following. If a functional CPT for variable X appears in both parts of a product, then variable X can be summed out from one part without changing the value of the product.

Theorem 3. *Consider a functional CPT f for variable X. If $f \in \mathcal{G}$ and $f \in \mathcal{H}$, then $\mathcal{G}\cdot\mathcal{H} = \mathcal{G}\sum_X \mathcal{H}$.*

Proof. Suppose CPT f is over variables $X\mathbf{U}$. Let $h(\mathbf{X})$ and $g(\mathbf{Y})$ be the factors corresponding to $\mathcal{H}$ and $\mathcal{G}$, respectively. Let $\mathbf{Z} = \mathbf{X}\cup\mathbf{Y}$ and $\mathbf{X}^\star = \mathbf{X}\backslash\{X\}$. Then variables $X\mathbf{U}$ must belong to $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$, and parents $\mathbf{U}$ must belong to $\mathbf{X}^\star$. Let $e_l = \mathcal{G}\cdot\mathcal{H}$ and $e_r = \mathcal{G}\sum_X\mathcal{H}$. We want to show $e_l(\mathbf{z}) = e_r(\mathbf{z})$ for every instantiation $\mathbf{z}$.

Consider an instantiation $\mathbf{z}$ and let $\mathbf{u}$, $\mathbf{x}^\star$, $\mathbf{x}$ and $\mathbf{y}$ be the instantiations of $\mathbf{U}$, $\mathbf{X}^\star$, $\mathbf{X}$ and $\mathbf{Y}$ in $\mathbf{z}$. Then $e_l(\mathbf{z}) = g(\mathbf{y})h(\mathbf{x})$ and $e_r(\mathbf{z}) = g(\mathbf{y})\sum_x h(x\mathbf{x}^\star)$. Since CPT f is functional, $f(x\mathbf{u}) \in \{0,1\}$ for any x and there is a unique x, call it $x_\mathbf{u}$, such that $f(x\mathbf{u}) = 1$.

If $f(x\mathbf{u}) = 0$, then $h(x\mathbf{x}^\star) = 0$ since $f\in\mathcal{H}$, leading to

$$e_r(\mathbf{z}) = g(\mathbf{y})\sum_x h(x\mathbf{x}^\star) = g(\mathbf{y})\sum_{\substack{x\\ f(x\mathbf{u})=1}} h(x\mathbf{x}^\star) = g(\mathbf{y})h(x_\mathbf{u}\mathbf{x}^\star).$$

If $x_\mathbf{u}$ is the instantiation of X in $\mathbf{z}$, then $x_\mathbf{u}\mathbf{x}^\star = \mathbf{x}$ and $e_r(\mathbf{z}) = g(\mathbf{y})h(\mathbf{x}) = e_l(\mathbf{z})$. Otherwise, $g(\mathbf{y}) = 0$ since $f\in\mathcal{G}$, which leads to $e_l(\mathbf{z}) = e_r(\mathbf{z}) = 0$. Hence, $e_l(\mathbf{z}) = e_r(\mathbf{z})$ for every instantiation $\mathbf{z}$ and we have $\mathcal{G}\cdot\mathcal{H} = \mathcal{G}\sum_X\mathcal{H}$. □

Theorem 3 has a key corollary. If a functional CPT for variable X appears in both parts of a product, we can sum out variable X from the product by independently summing it out from each part.

Corollary 1. *Consider a functional CPT f for variable X. If $f\in\mathcal{G}$ and $f\in\mathcal{H}$, then $\sum_X \mathcal{G}\cdot\mathcal{H} = \left(\sum_X\mathcal{G}\right)\left(\sum_X\mathcal{H}\right)$.*

Proof. $\sum_X\mathcal{G}\cdot\mathcal{H} = \sum_X\left(\mathcal{G}\sum_X\mathcal{H}\right)$ by Theorem 3, which equals $\left(\sum_X\mathcal{H}\right)\left(\sum_X\mathcal{G}\right)$ by Theorem 2. □

Theorem 3 and Corollary 1 may appear unusable as they require multiple occurrences of a functional CPT whereas the factors of a Bayesian network contain a single (functional) CPT for each variable. This is where the second result comes in: *duplicating* a functional CPT in a product of factors does not change the product value.

Theorem 4. *For functional CPT f, if $f\in\mathcal{G}$, then $f\cdot\mathcal{G} = \mathcal{G}$.*

Proof. Let $g(\mathbf{Z})$ be the product of factors in $\mathcal{G}$ and let $h = f\cdot g$. Suppose factor f is the CPT of variable X and parents $\mathbf{U}$. Consider an instantiation $\mathbf{z}$ and suppose it includes instantiation $x\mathbf{u}$. If $f(x\mathbf{u}) = 0$, then $g(\mathbf{z}) = 0$ since $f\in\mathcal{G}$. Moreover, $h(\mathbf{z}) = f(x\mathbf{u})g(\mathbf{z}) = 0$. If $f(x\mathbf{u}) = 1$, then $h(\mathbf{z}) = f(x\mathbf{u})g(\mathbf{z}) = g(\mathbf{z})$. Hence, $g(\mathbf{z}) = h(\mathbf{z})$ for all instantiations $\mathbf{z}$ and we have $\mathcal{G} = f\cdot\mathcal{G}$. □

Theorem 4 holds if f embeds any functional dependency that is implied by factors $\mathcal{G}$ instead of being a functional CPT in $\mathcal{G}$ but we do not pursue the applications of this generalization in this paper.

To see how Theorems 3 and 4 interplay, consider the f-expression $\sum_X f(XY)g(XZ)h(XW)$. In the standard VE algorithm, one must multiply all three factors before summing out variable X, leading to a factor over four variables $XYZW$. If factor f is

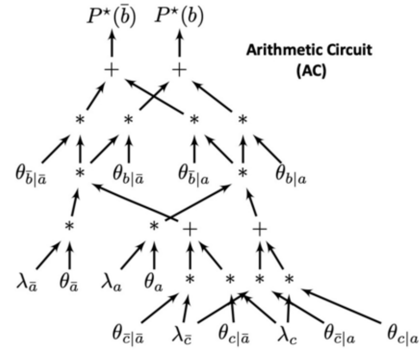

Figure 1: An arithmetic circuit (AC) compiled from the Bayesian network $A\rightarrow B$, $A\rightarrow C$. The AC computes factor $f(B)$, where ηf is the posterior on variable B given evidence on variables A and C.

a functional CPT for variable X, we can duplicate it by Theorem 4: $f(XY)g(XZ)h(XW) = f(XY)g(XZ)f(XY)h(XW)$. Moreover, Corollary 1 gives $\sum_X f(XY)g(XZ)f(XY)h(XW) = \sum_X f(XY)g(XZ)\sum_X f(XY)h(XW)$, which avoids constructing a factor over four variables. We show in Section 3 how these theorems enable efficient inference on models with very large treewidth.

3 Variable Elimination with Functional CPTs

We now present our proposed VE algorithm. We first present a standard VE algorithm based on *jointrees* [18] and then extend it to exploit functional CPTs. Our algorithm will not compute probabilities, but will compile *symbolic f-expressions* whose factors contain *symbolic parameters.* A symbolic f-expression is compiled once and used thereafter to answer multiple queries. Moreover, its parameters can be learned from labeled data using gradient descent. We will show how to map symbolic f-expressions into tensor graphs in Section 4 and use these graphs for supervised learning in Section 5.

Once the factors of a symbolic f-expression are unfolded, the result is an Arithmetic Circuits (ACs) [9, 4] as shown in Figure 1. In fact, the standard VE algorithm we present next is a refinement on the one proposed in [9] for extracting ACs from jointrees.

The next section introduces jointrees and some key concepts that we need for the standard and extended VE algorithms.

3.1 Jointrees

Consider the Bayesian network in the middle of Figure 2 and its jointree on the left of the figure. The jointree is simply a tree with factors attached to some of its nodes (the circles in Figure 2 are the jointree nodes). We use *binary jointrees* [28], in which each node has either one or three neighbors and where nodes with a single neighbor are called *leaves.* The two jointrees in Figure 2 are identical but arranged differently. The one on the left has leaf node 2 at the top and the one on the right has leaf node 3 at the top.

Our use of jointrees deviates from normal for reasons that become apparent later. First, we use a binary jointree whose leaves are in one-to-one correspondence with model variables. Second, we only attach factors to leaf nodes: The CPT and evidence indicator for each variable X are assigned to the leaf node i corresponding to variable X. Leaf jointree node i is called the *host* of variable X in this case.[5]

[5] For similar uses and a method for constructing such binary jointrees, see [7] and [10, Chapter 8]. *Contraction trees* which were adopted later for contracting tensor networks [15] correspond to binary jointrees.

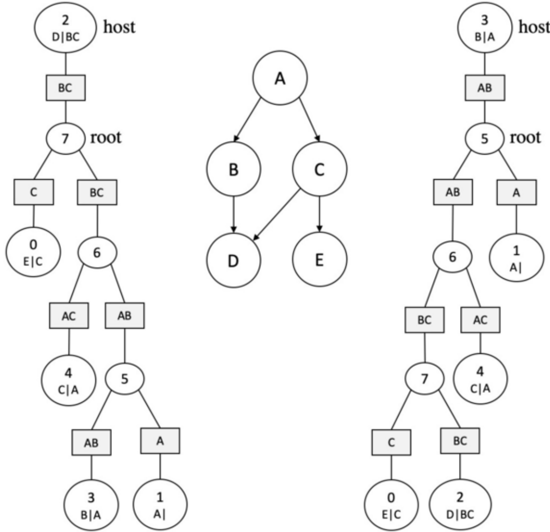

Figure 2: A Bayesian network with a jointree (two views).

The Bayesian network in Figure 2 has five variables. Its jointree also has five leaves, each of which hosts a network variable. For example, jointree node 2 at the top-left hosts variable D: the CPT and evidence indicator for variable D are assigned to this jointree node.

A key notion underlying jointrees are *edge separators* which determine the space complexity of inference (the rectangles in Figure 2 are separators). The separator for edge (i, j), denoted $\mathtt{sep}(i, j)$, are model variables that appear in leaf nodes on both sides of the edge. For example, $\mathtt{sep}(6, 7) = \{B, C\}$ as these are the model variables that appear in jointree leaves $\{0, 2\}$ and $\{1, 3, 4\}$. A related notion is the *cluster* of jointree node i. If i is leaf, its cluster are the variables appearing at node i. Otherwise, it is the union of separators for edges (i, j). Every factor constructed by VE is over the variables of some separator or cluster. The time complexity of VE is exponential in the size of clusters and linear in the number of nodes in a jointree.

The size of largest cluster -1 is called the *jointree width* and cannot be lower than the Bayesian network treewidth; see [10, Chapter 9] for a detailed treatment of this subject. When the network contains variables with different cardinalities, the size of a cluster is better measured by the number of instantiations that its variables has. We therefore define the *binary rank* of a cluster as *log2* of its instantiation count. The binary rank coincides with the number of variables in a cluster when all variables are binary.

Our technique for exploiting functional dependencies will use Theorems 3 and 4 to shrink the size of clusters and separators significantly below jointree width, allowing us to handle networks with very large treewidth. The algorithm will basically reduce the maximum binary rank of clusters and separators, which can exponentially reduce the size of factors constructed by VE during inference.

3.2 Compiling Symbolic f-expressions using VE

Suppose we wish to compile an f-expression that computes the posterior on variable Q. We first identify the leaf jointree node h that hosts variable Q. We then arrange the jointree so *host* h is at the top as in Figure 2. Host h will then have a single child r which we call the jointree *root*. The tree rooted at node r is now a binary tree, with each node i having two children c_1 and c_2 and a parent p. On the left of Figure 2, root r=7 has two children c_1=0, c_2=6 and parent p=2. We refer to such a jointree arrangement as a *jointree view.*

Jointree views simplify notation. For example, we can now write $\mathtt{sep}(i)$ to denote the separator between node i and its parent p instead of $\mathtt{sep}(i, p)$. We will adopt this simpler notation from now on.

We now compile an f-expression using the following equations:

$$P^\star(Q) \quad = \quad \eta \overset{=}{\sum_{Q}} \mathcal{F}_h f(r) \tag{1}$$

$$f(i) \quad = \quad \begin{cases} \overset{=}{\sum\limits_{\mathtt{sep}(i)}} \mathcal{F}_i & i \text{ is leaf} \\ \overset{=}{\sum\limits_{\mathtt{sep}(i)}} f(c_1) f(c_2) & i \text{ has children } c_1, c_2 \end{cases} \tag{2}$$

Here, $\mathcal{F}_i$ is the product of factors assigned to leaf node i (CPT and evidence indicator for the model variable assigned to node i).

For the jointree view in Figure 2 (left), applying these equations to variable Q=D, host h=2 and root r=7 yields the f-expression:

$$P^\star(D) = \eta \overset{=}{\sum_{D}} \mathcal{F}_2 \overset{=}{\sum_{BC}} [\overset{=}{\sum_{C}} \mathcal{F}_0][\overset{=}{\sum_{BC}}[\overset{=}{\sum_{AC}} \mathcal{F}_4][\overset{=}{\sum_{AB}}[\overset{=}{\sum_{AB}} \mathcal{F}_3][\overset{=}{\sum_{A}} \mathcal{F}_1]]].$$

This expression results from applying Equation 1 to the host h=2 followed by applying Equation 2 to each edge in the jointree. Each sum in the expression corresponds to a separator and every product constructed by the expression will be over the variables of a cluster.

Our compiled AC is simply the above f-expression. The value of the expression represents the circuit output. The evidence indicators in the expression represent the circuit inputs. Finally, the CPTs of the expression contain the circuit parameters (see the AC in Figure 1).

We will now introduce new notation to explain Equations 1 and 2 as we need this understanding in the following section; see also [10, Chapter 7]. For node i in a jointree view, we use $\breve{\mathcal{F}}_i$ to denote the set of factors at or below node i. We also use $\widehat{\mathcal{F}}_i$ to denote the set of factors above node i. Consider node 6 on the left of Figure 2. Then $\breve{\mathcal{F}}_6$ contains the factors assigned to leaf nodes $\{1, 3, 4\}$ and $\widehat{\mathcal{F}}_6$ contains the factors assigned to leaf nodes $\{0, 2\}$.

For a jointree view with host h and root r, $\widehat{\mathcal{F}}_r \breve{\mathcal{F}}_r$ contains all factors in the jointree and $\widehat{\mathcal{F}}_r = \mathcal{F}_h$. Equation 1 computes $\eta \overset{=}{\sum}_{Q} \widehat{\mathcal{F}}_r \breve{\mathcal{F}}_r$, while delegating the computation of product $\breve{\mathcal{F}}_r$ to Equation 2, which actually computes $\overset{=}{\sum}_{\mathtt{sep}(r)} \breve{\mathcal{F}}_r$ by summing out all variables but for ones in $\mathtt{sep}(r)$. The equation uses the decomposition $\breve{\mathcal{F}}_i = \breve{\mathcal{F}}_{c_1} \breve{\mathcal{F}}_{c_2}$ to sum out variables more aggressively:

$$\begin{aligned} f(i) \quad &= \quad \overset{=}{\sum_{\mathtt{sep}(i)}} \breve{\mathcal{F}}_i = \overset{=}{\sum_{\mathtt{sep}(i)}} \breve{\mathcal{F}}_{c_1} \breve{\mathcal{F}}_{c_2} \\ &= \quad \overset{=}{\sum_{\mathtt{sep}(i)}} \left(\overset{=}{\sum_{\mathtt{sep}(c_1)}} \breve{\mathcal{F}}_{c_1} \right) \left(\overset{=}{\sum_{\mathtt{sep}(c_2)}} \breve{\mathcal{F}}_{c_2} \right). \end{aligned} \tag{3}$$

The rule employed by Equation 2 is simple: sum out from product $\breve{\mathcal{F}}_i$ all variables except ones appearing in product $\widehat{\mathcal{F}}_i$ (Theorem 2). The only variables shared between factors $\breve{\mathcal{F}}_i$ and $\widehat{\mathcal{F}}_i$ are the ones in $\mathtt{sep}(i)$ so Equation 2 is exploiting Theorem 2 to the max. The earlier that variables are summed out, the smaller the factors we need to multiply and the smaller the f-expressions that VE compiles.

3.3 Exploiting Functional Dependencies

We now present an algorithm that uses Theorems 3 and 4 to sum out variables earlier than is licensed by Theorems 1 and 2. Here, 'earlier' means lower in the jointree view which leads to smaller factors.

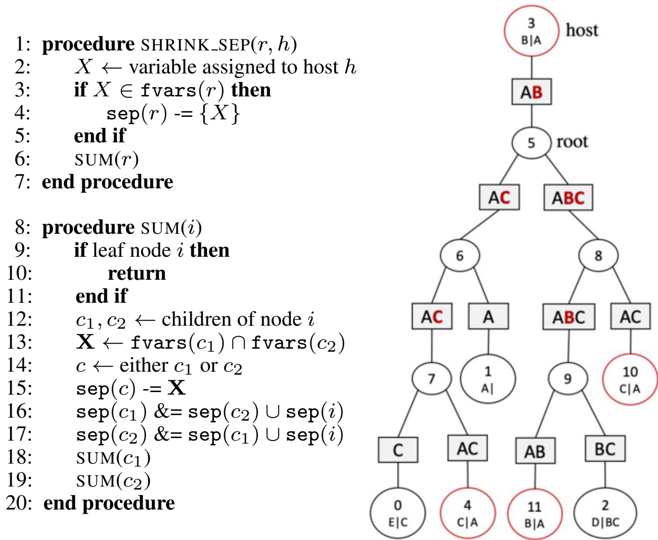

Figure 3: **Left:** Algorithm for shrinking separators based on functional CPTs. **Right:** An application of the algorithm where dropped variables are colored red. Variables B and C have functional CPTs.

Our algorithm uses the notation $\texttt{fvars}(i)$ to denote the set of variables that have a functional CPT at or below node i in the jointree view. For example, in Figure 3, we have $\texttt{fvars}(8) = \{B, C\}$, $\texttt{fvars}(11) = \{B\}$ and $\texttt{fvars}(2) = \{\}$.

The algorithm is depicted in Figure 3 and is a direct application of Theorem 3 with a few subtleties. The algorithm traverses the jointree view top-down, removing variables from the separators of visited nodes. It is called on root r and host h of the view, SHRINK_SEP(r, h). It first shrinks the separator of root r which decomposes the set of factors into $\widehat{\mathcal{F}}_r\breve{\mathcal{F}}_r$. The only functional CPT that can be shared between factors $\widehat{\mathcal{F}}_r$ and $\breve{\mathcal{F}}_r$ is the one for variable X assigned to host h. If variable X is functional and its CPT is shared, Theorem 3 immediately gives $\widehat{\mathcal{F}}_r\breve{\mathcal{F}}_r = \widehat{\mathcal{F}}_r \sum_X \breve{\mathcal{F}}_r$. Variable X can then be summed at root r by dropping it from $\texttt{sep}(r)$ as done on line 4.

The algorithm then recurses on the children of root r. The algorithm processes both children c_1 and c_2 of a node before it recurses on these children. This is critical as we explain later. The set $\mathbf{X}$ computed on line 13 contains variables that have functional CPTs in both factors $\breve{\mathcal{F}}_{c_1}$ and factors $\breve{\mathcal{F}}_{c_2}$ (recall Equation 3). Theorem 3 allows us to sum out these variables from either $\breve{\mathcal{F}}_{c_1}$ or $\breve{\mathcal{F}}_{c_2}$ but not both, a choice that is made on line 14. A variable that has a functional CPT in both $\breve{\mathcal{F}}_{c_1}$ and $\breve{\mathcal{F}}_{c_2}$ is summed out from one of them by dropping it from either $\texttt{sep}(c_1)$ or $\texttt{sep}(c_2)$ on line 15. In our implementation, we heuristically choose a child based on the size of separators below it. We add the sizes of these separators (number of instantiations) and choose the child with the largest size breaking ties arbitrarily.

If a variable is summed out at node i and at its child c_2, we can sum it out earlier at child c_1 by Theorem 2 (classical VE): $\sum_X(\breve{\mathcal{F}}_{c_1} \sum_X \breve{\mathcal{F}}_{c_2}) = (\sum_X \breve{\mathcal{F}}_{c_1})(\sum_X \breve{\mathcal{F}}_{c_2})$. A symmetric situation arises for child c_2. This is handled on lines 16-17. Applying Theorem 2 in this context demands that we process nodes c_1 and c_2 before we process their children. Otherwise, the reduction of separators $\texttt{sep}(c_1)$ and $\texttt{sep}(c_2)$ will not propagate downwards early enough, missing opportunities for applying Theorem 2 further.

Figure 3 depicts an example of applying algorithm SHRINK_SEP to a jointree view for the Bayesian network in Figure 2. Variables colored red are dropped by SHRINK_SEP. The algorithm starts by processing root $r = 5$, dropping variable B from $\texttt{sep}(5)$ on line 4. It then processes children $c_1 = 6$ and $c_2 = 8$ simultaneously. Since both children contain a functional CPT for variable C, the variable can be dropped from either $\texttt{sep}(6)$ or $\texttt{sep}(8)$. Child $c_2 = 8$ is chosen in this case and variable C is dropped from $\texttt{sep}(8)$. We have $\texttt{sep}(6) = \{A, C\}$ and $\texttt{sep}(8) = \{A, B\}$ at this point. Lines 16-17 shrink these separators further to $\texttt{sep}(6) = \{A\}$ and $\texttt{sep}(8) = \{A\}$.

Our proposed technique for shrinking separators will have an effect only when functional CPTs have multiple occurrences in a jointree (otherwise, set $\mathbf{X}$ on line 13 is always empty). While this deviates from the standard use of jointrees, replicating functional CPTs is licensed by Theorem 4. The (heuristic) approach we adopted for replicating functional CPTs in a jointree is based on replicating them in the Bayesian network. Suppose variable X has a functional CPT and children $C_1, \ldots, C_n$ in the network, where $n > 1$. We replace variable X with replicas $X_1, \ldots, X_n$. Each replica X_i has a single child C_i and the same parents as X. We then construct a jointree for the resulting network and finally replace each replica X_i by X in the jointree. This creates n replicas of the functional CPT in the jointree. Replicating functional CPTs leads to jointrees with more nodes, but smaller separators and clusters as we shall see in Section 5.

4 Mapping ACs into Tensor Graphs

We discuss next how we map ACs (symbolic f-expressions) into tensors graphs for efficient inference and learning. Our implementation is part of the PYTAC system under development by the author. PYTAC is built on top of TensorFlow and will be open sourced.

A *tensor* is a data structure for a multi-dimensional array. The *shape* of a tensor defines the array dimensions. A tensor with shape $(2, 2, 3)$ has $2 \times 2 \times 3$ *elements* or *entries.* The dimensions of a tensor are numbered and called *axes.* The number of axes is the tensor *rank.* Tensor computations can be organized into a *tensor graph:* a data flow graph with nodes representing tensor operations. Tensors form the basis of many machine learning tools today.

A factor over variables $X_1, \ldots, X_n$ can be represented by a tensor with rank n and shape $(d_1, \ldots, d_n)$, where d_i is the cardinality of variable X_i (i.e., its number of values). Factor operations can then be implemented using tensor operations, leading to a few advantages. First, tensor operations are heavily optimized to take advantage of special instruction sets and architectures (on CPUs and GPUs) so they can be orders of magnitude faster than standard implementations of factor operations (even on laptops). Next, the elements of a tensor can be variables, allowing one to represent symbolic f-expressions, which is essential for mapping ACs into tensor graphs that can be trained. Finally, tools such as TensorFlow and PyTorch provide support for computing the partial derivates of a tensor graph with respect to tensor elements, and come with effective gradient descent algorithms for optimizing tensor graphs (and hence ACs). This is very useful for training ACs from labeled data as we do in Section 5.

To map ACs (symbolic f-expressions) into tensor graphs, we need to implement factor multiplication, summation and normalization. Mapping factor summation and normalization into tensor operations is straightforward: summation has a corresponding tensor operation (TF.REDUCE_SUM) and normalization can be implemented using tensor summation and division. Factor multiplication does not have a corresponding tensor operation and leads to some complications.[6]

[6] Tensor multiplication is pointwise while factors are normally over different sets of variables. Hence, multiplying the tensors corresponding to factors $f(ABC)$ and $g(BDE)$ does not yield the expected result. The simplest option is to use TF.EINSUM, which can perform factor multiplication if we pass it the string "abc, bde –> abcde" (`https://www.tensorflow.`

We bypassed these complications in the process of achieving something more ambitious. Consider Equation 2 which contains almost all multiplications performed by VE. Factors $f_1(c_1)$, $f_2(c_2)$ and the result $f(i)$ are over separators $\mathtt{sep}(c_1)$, $\mathtt{sep}(c_2)$ and $\mathtt{sep}(i)$. This equation multiplies factors f_1 and f_2 to yield a factor over variables $\mathtt{sep}(c_1) \cup \mathtt{sep}(c_2)$ and then shrinks it by summation into a factor over variables $\mathtt{sep}(i)$. We wanted to avoid constructing the larger factor before shrinking it. That is, we wanted to multiply-then-sum in one shot as this can reduce the size of our tensor graphs significantly.[7] A key observation allows this using standard tensor operations.

The previous separators are all connected to jointree node i so they satisfy the following property [10, Chapter 9]: If a variable appears in one separator, it also appears in at least one other separator. Variables $\mathtt{sep}(c_1) \cup \mathtt{sep}(c_2) \cup \mathtt{sep}(i)$ can then be partitioned as follows:[8]

$\mathbf{C}$: variables in f_1, f_2 and f, $\mathtt{sep}(c_1) \cap \mathtt{sep}(c_2) \cap \mathtt{sep}(i)$
$\mathbf{X}$: variables in f_1, f but not f_2, $(\mathtt{sep}(c_1) \cap \mathtt{sep}(i)) \setminus \mathtt{sep}(c_2)$
$\mathbf{Y}$: variables in f_2, f but not f_1, $(\mathtt{sep}(c_2) \cap \mathtt{sep}(i)) \setminus \mathtt{sep}(c_1)$
$\mathbf{S}$: variables in f_1, f_2 but not f, $(\mathtt{sep}(c_1) \cap \mathtt{sep}(c_2)) \setminus \mathtt{sep}(i)$

where variables $\mathbf{S}$ are the ones summed out by Equation 2. The variables in each factor can now be structured as follows: $f_1(\mathbf{C}, \mathbf{X}, \mathbf{S})$, $f_2(\mathbf{C}, \mathbf{Y}, \mathbf{S})$ and $f(\mathbf{C}, \mathbf{X}, \mathbf{Y})$. We actually group each set of variables $\mathbf{C}, \mathbf{X}, \mathbf{Y}$ and $\mathbf{S}$ into a single compound variable so that factors f_1, f_2 and f can each be represented by a rank-3 tensor. We then use the tensor operation for matrix multiplication TF.MATMUL to compute $f = \sum_{\mathbf{S}} f_1 f_2$ in one shot, without having to construct a tensor for the product $f_1 f_2$. Matrix multiplication is perhaps one of the most optimized tensor operations on both CPUs and GPUs.

Preparing tensors $f_1(\mathbf{C}, \mathbf{X}, \mathbf{S})$ and $f_2(\mathbf{C}, \mathbf{Y}, \mathbf{S})$ for matrix multiplication requires two operations: TF.RESHAPE which aggregate variables into compound dimensions and TF.TRANSPOSE which order the resulting dimensions so TF.MATMUL can map f_1 and f_2 into $f(\mathbf{C}, \mathbf{X}, \mathbf{Y})$. The common dimension $\mathbf{C}$ must appear first in f_1 and f_2. Moreover, the last two dimensions must be ordered as $(\mathbf{X}, \mathbf{S})$ and $(\mathbf{S}, \mathbf{Y})$ but TF.MATMUL can transpose the last two dimensions of an input tensor on the fly if needed. Using matrix multiplication in this fashion had a significant impact on reducing the size of tensor graphs and the efficiency of evaluating them, despite the added expense of using TF.TRANSPOSE and TF.RESHAPE operations (the latter operation does not use space and is very efficient).

PYTAC represents ACs using an *abstract* tensor graph called an *ops graph,* which can be mapped into a particular tensor implementation depending on the used machine learning tool. PYTAC also has a dimension management utility, which associates each tensor with its structured dimensions while ensuring that all tensors are structured appropriately so operations can be applied to them efficiently. We currently map an ops graph into a TF.GRAPH object, using the TF.FUNCTION utility introduced recently in TensorFlow 2.0.0. PYTAC also supports the recently introduced *Testing Arithmetic Circuits (TACs),* which augment ACs with *testing units* that turns them into universal function approximators like neural networks [6, 5, 27].

org/api_docs/python/tf/einsum). We found this too inefficient though for extensive use as it performs too many tensor transpositions. One can also use the technique of *broadcasting* by adding *trivial* dimensions to align tensors (https://www.tensorflow.org/xla/broadcasting), but broadcasting has limited support in TensorFlow requiring tensors with small enough ranks.

[7] See a discussion of this space issue in [10, Chapter 7].

[8] In a jointree, every separator that is connected to a node is a subset of the union of other separators connected to that node. Hence, $\mathtt{sep}(i) \subseteq \mathtt{sep}(c_1) \cup \mathtt{sep}(c_2)$.

5 Case Studies

We next evaluate the proposed VE algorithm on two classes of models that have abundant functional dependencies. We also evaluate the algorithm on randomly generated Bayesian networks while varying the amount of functional dependencies. The binary jointrees constructed for these models are very large and prohibit inference using standard VE. We constructed these binary jointrees from variable elimination orders using the method proposed in [7]; see also [10, Chapter 9]. The elimination orders were obtained by the *minfill heuristic;* see, e.g., [20].[9]

5.1 Rectangle Model

We first consider a generative model for rectangles shown in Figure 4. In an image of size $n \times n$, a rectangle is defined by its upper left corner (row, col), height and width. Each of these variables has n values. The rectangle also has a binary label variable, which is either tall or wide. Each row has a binary variable $\mathtt{row}_i$ indicating whether the rectangle will render in that row (n variables total). Each column has a similar variable $\mathtt{col}_j$. We also have n^2 binary variables which correspond to image pixels ($\mathtt{pixel}_{ij}$) indicating whether the pixel is on or off. This model can be used to predict rectangle attributes from noisy images such as those shown in Figure 4. We use the model to predict whether a rectangle is tall or wide by compiling an AC with variable label as output and variables $\mathtt{pixel}_{ij}$ as input. The AC computes a distribution on label given a noisy image as evidence and can be trained from labeled data using cross entropy as the loss function.[10]

Our focus is on the variables $\mathtt{row}_i$ and $\mathtt{col}_j$ which are determined by row/height and col/width, respectively (for example, $\mathtt{row}_i$ is on iff $\mathtt{row} \leq i < \mathtt{row}+\ \mathtt{height}$). In particular, we will investigate the impact of these functional relationships on the efficiency of our VE compilation algorithm and their impact on learning AC parameters from labeled data. Our experiments were run on a MacBook Pro, 2.2 GHz Intel Core i7, with 32 GB RAM.

Table 1 depicts statistics on ACs that we compiled using our proposed VE algorithm. For each image size, we compiled an AC for predicting the rectangle label while exploiting functional CPTs to remove variables from separators during the compilation process. As shown in the table, exploiting functional CPTs has a dramatic impact on the complexity of VE. This is indicated by the size of largest jointree cluster (binary rank) in a classical jointree vs one whose separators and clusters where shrunk due to functional dependencies.[11] Recall that a factor over a cluster will have a size exponential in the cluster binary rank (the same for factors over separators). The table also shows the size of compiled ACs, which is the sum of tensor sizes in the corresponding tensor graph (the tensor size is the number of elements/entries it has). For a baseline, the AC obtained by standard

[9] The minfill heuristic and similar ones aim for jointrees that minimize the size of largest cluster (i.e., treewidth). It was observed recently that minimizing the size of largest separator (called *max rank*) is more desirable when using tensors since the memory requirements of Equation 2 can depend only on the size of separators not clusters (see [14] for recent methods that optimize max rank). This observation holds even when using classical implementations of the jointree algorithm and was exploited earlier to reduce the memory requirements of jointree inference; see, e.g., [22, 13].

[10] Arthur Choi suggested the use of rectangle models and Haiying Huang proposed this particular version of the model.

[11] We applied standard node and value pruning to the Bayesian network before computing a jointree and shrinking it. This has more effect on the digits model in Section 5.2. For example, it can infer that some pixels will never be turned on as they will never be occupied by any digit.

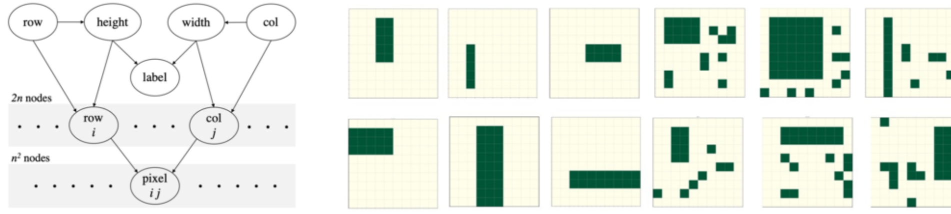

Figure 4: **Left:** A generative model for rectangles. **Right:** Examples of clean and noisy rectangle images.

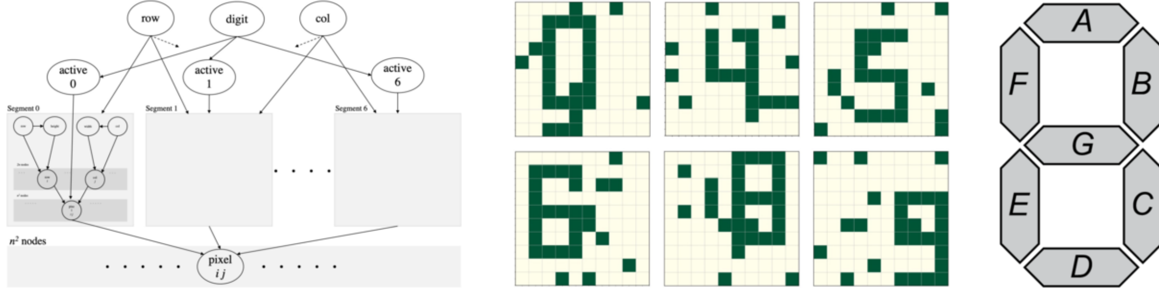

Figure 5: **Left:** A generative model for seven-segment digits. **Middle:** Examples of noisy digit images. **Right:** Seven-segment digit.

VE (without exploiting functional CPTs) for an image of size 20×20 is $18,032,742,365$, which is about 80 times larger than the size of AC reported in Table 1. What is particularly impressive is the time it takes to evaluate these ACs (compute their output from input). On average it takes about 7 milliseconds to evaluate an AC of size ten million for these models, which shows the promise tensor-based implementations (these experiments were run on a laptop).

We next investigate the impact of integrating background knowledge when learning AC parameters. For training, we generated labeled data for all clean images of rectangles and added n noisy images for each (with the same label). Noise is generated by randomly flipping $min(n, a-1, b/2)$ background pixels, where a is the number of rectangle pixels and b is the number of background pixels. We used the same process for testing data, except that we increased the number of noisy pixels to $min(2*n, a-1, b/2)$ and doubled the number of noisy images. We trained the AC using cross entropy as the loss function to minimize the classification accuracy.[12]

Table 2 shows the accuracy of classifying rectangles (tall vs wide) on 10×10 images using ACs with and without background knowledge. ACs compiled from models with background knowledge have fewer parameters and therefore need less data to train. The training and testing examples were selected randomly from the datasets described above with 1000 examples always used for testing, regardless of the training data size. Each classification accuracy is the average over twenty five runs. The table clearly shows that integrating background knowledge into the compiled AC yields higher classification accuracies given a fixed number of training examples.

5.2 Digits Model

We next consider a generative model for seven-segment digits shown in Figure 5 (`https://en.wikipedia.org/wiki/Seven-segment_display`). The main goal of this model is to recognize digits in noisy images such as those shown in Figure 5. The model has four vertical and three horizontal segments. A digit is generated by activating some of the segments. For example, digit

[12] Some of the CPTs contain zero parameters but are not functional, such as the ones for `width` and `height`. We fixed these zeros in the AC when learning with background knowledge. We also tied the parameters of the $\texttt{pixel}_{ij}$ variables therefore learning one CPT for all of them.

Table 1: Size and compile/evaluation time for ACs that compute the posterior on rectangle label. Reported times are in seconds. Evaluation time is the average of evaluating an AC over a batch of examples.

Image Size	Functional CPTs	Network Nodes	Max Cluster Size		AC Size	**Eval Time**	Compile Time
			rank	binary rank			
8×8	✗	85	11	15.0	$926,778$	.001	4.9
	✓	197	5	**13.0**			
10×10	✗	125	13	17.6	$3,518,848$	.003	2.9
	✓	305	5	**14.3**			
12×12	✗	173	15	20.2	$10,485,538$	.007	4.1
	✓	437	5	**15.3**			
14×14	✗	229	17	22.6	$26,412,192$	.018	5.7
	✓	593	5	**16.2**			
16×16	✗	293	19	25.0	$58,814,458$	.034	7.4
	✓	773	5	**17.0**			
20×20	✗	445	23	29.6	$224,211,138$	.140	14.1
	✓	1205	5	**18.3**			

Table 2: Classification accuracy on 10×10 noisy rectangle images. Testing data included 1000 examples in each case.

Functional CPTs	Accuracy	Number of Training Examples						Param Count
		25	50	100	250	500	1000	
fixed in AC	mean	82.64	89.16	96.08	97.92	99.51	98.39	136
	stdev	15.06	11.98	8.34	5.56	0.62	7.00	
trainable	mean	53.29	56.92	62.20	74.62	84.94	88.69	$4,428$
	stdev	1.89	5.31	6.95	5.29	3.14	2.79	

Table 3: Size and compile/evaluation time for ACs that compute a posterior over digits. Reported times are in seconds. Evaluation time is the average of evaluating an AC over a batch of examples.

Image Size	Functional CPTs	Network Nodes	Max Cluster Size		AC Size	**Eval Time**	Compile Time
			rank	binary rank			
8×8	✗	638	32	33.3	$264,357$	.008	9.3
	✓	1155	9	**12.6**			
10×10	✗	954	59	60.8	$2,241,205$	.008	13.6
	✓	2173	9	**14.1**			
12×12	✗	1334	81	83.8	$11,625,558$	.014	23.2
	✓	3469	10	**16.7**			
14×14	✗	1778	116	121.0	$32,057,227$	.030	36.8
	✓	5007	11	**18.4**			
16×16	✗	2286	134	140.0	$95,094,167$	.076	50.4
	✓	6825	11	**19.3**			

Table 4: Classification accuracy on 10×10 noisy digit images. Testing data included 1000 examples in each case.

Functional CPTs	Accuracy	Number of Training Examples						Param Count
		25	50	100	250	500	1000	
fixed in AC	mean	83.51	89.17	94.94	97.68	98.49	98.44	275
	stdev	8.70	6.02	4.57	1.45	0.91	0.27	
trainable	mean	9.82	12.26	13.28	22.36	29.51	35.67	$22,797$
	stdev	0.77	2.25	3.32	3.45	2.40	1.57	

8 is generated by activating all segments and digit 1 by activating two vertical segments. Segments are represented by rectangles as in the previous section, so this model integrates seven rectangle models. A digit has a location specified by the row and column of its upper-left corner (height is seven pixels and width is four pixels). Moreover, each segment has an activation node which is turned on or off depending on the digit. When this activation node is off, segment pixels are also turned off. An image of size $n \times n$ has n^2 pixels whose values are determined by the pixels generated by segments.

This is a much more complex and larger model than the rectangle model and also has an abundance of functional dependencies. It is also much more challenging computationally. This can be seen by examining Tables 3, which reports the size of largest clusters in the jointrees for this model. For example, the model for 16×16 images has a cluster with a binary rank of 140. This means that standard VE would have to construct a factor of size 2^{140} which is impossible. Our proposed technique for exploiting functional dependencies makes this possible though as it reduces the binary rank of largest cluster down to 19.3. And even though the corresponding AC has size of about one hundred million, it can be evaluated in about 76 milliseconds. The AC compilation times are also relatively modest.

We trained the compiled ACs as we did in the previous section. We generated all clean images and added noise as follows. For each clean image we added 100 noisy images for training and 200 for testing by randomly flipping n background pixels where n is the image size.

Table 4 parallels the one for the rectangle model. We trained two ACs, one that integrates background knowledge and one that does not. The former AC has fewer parameters and therefore requires less data to train. While this is expected, it is still interesting to see how little data one needs to get reasonable accuracies. In general, Tables 3 and 4 reveal the same patterns of the rectangle model: exploiting functional dependencies leads to a dramatic reduction in the AC size and integrating background knowledge into the compiled AC significantly improves learnability.

5.3 Random Bayesian Networks

Table 5: Reduction in maximum cluster size due to exploiting functional dependencies. The number of values a node has was chosen randomly from $(2, 3)$. We averaged over 10 random networks for each combination of network node count, maximal parent count and the percentage of nodes having functional CPTs. The parents of a node and their count where chosen randomly. Functional nodes where chosen randomly from non-root nodes. The *binary rank* of a cluster is *log2* of the number of its instantiations.

Network Node Count	Maximal Parent Count	Percentage Functional Nodes %	**Binary Rank of Largest Cluster**					
			Original Jointree		Shrunk Jointree		Reduction	
			mean	stdev	mean	stdev	**mean**	stdev
75	4	25	22.4	2.8	19.4	3.1	**3.0**	1.7
		50	22.5	2.2	16.9	1.8	**5.6**	2.5
		67	22.9	3.9	13.1	2.3	**9.8**	3.4
		80	21.9	2.7	11.1	1.9	**10.8**	3.2
100	5	25	38.7	4.5	33.1	4.6	**5.7**	2.0
		50	38.1	2.9	23.7	3.3	**14.4**	4.3
		67	38.0	3.2	18.9	3.1	**19.1**	3.9
		80	36.8	3.0	13.5	2.5	**23.3**	3.1
150	6	25	64.3	5.4	54.2	4.4	**10.1**	4.2
		50	64.9	3.2	41.9	5.6	**23.0**	5.1
		67	64.3	6.0	28.2	4.2	**36.0**	4.7
		80	66.4	4.8	21.3	4.6	**45.1**	2.1

We next present two experiments on randomly generated Bayesian networks. The first experiment further evaluates our proposed algorithm for exploiting functional dependencies. The second experiment

Table 6: Comparing evaluation time of three AC representations: Tensor graph (**TenG**), scalar graph (**ScaG**) and scalar-batch graph (**ScaBaG**). We averaged over 10 random Bayesian networks for each combination of batch size and limit on circuit size. AC size limit is in millions of nodes. The binary rank of a tensor is *log2* of the number of its entries. Maximum binary rank is for the largest tensor in the tensor graph. Normalized time (tensor graph) is evaluation time per one million AC nodes (a node is a tensor entry). Each cell below contains the **mean** (top) and **stdev** (bottom). Times are in milliseconds.

Batch Size	Tensor Graph (TenG)			**Milliseconds**	Slow Down Factor	
	Limit on Size	Actual Size	Max Binary Rank	**TenG Time Normalized**	ScaG / TenG Time Ratio	ScaBaG / TenG Time Ratio
1	5-10 M	6,992,414 1,830,909	19.7 0.7	66.6 20.6	11.5 4.6	47.0 20.3
	15-20 M	17,979,799 1,391,918	21.2 0.5	34.3 3.0	22.2 7.1	82.1 23.7
	25-30 M	26,540,961 1,154,660	21.6 0.5	20.8 4.6	38.4 14.6	137.3 56.1
	35-40 M	37,058,914 1,349,479	21.8 0.4	16.0 3.5	50.3 32.1	177.2 128.7
10	5-10 M	8,157,025 1,599,408	20.0 0.5	7.8 2.3	112.3 45.9	38.0 24.8
	15-20 M	17,504,179 1,482,496	20.7 0.5	4.6 1.3	148.0 64.9	54.2 33.3
	25-30 M	27,728,478 2,029,237	21.7 0.9	4.5 1.3	209.7 51.0	60.1 17.0
	35-40 M	37,850,485 1,547,389	22.1 0.6	4.0 1.1	244.0 95.4	70.0 26.3
20	5-10 M	6,506,125 860,631	19.6 0.7	4.9 1.9	135.3 42.2	26.9 11.1
	15-20 M	17,766,240 1,209,040	20.7 0.5	3.1 1.3	251.5 123.7	39.9 15.2
	25-30 M	27,762,672 1,148,761	21.7 0.5	3.1 1.1	271.9 92.4	46.0 17.7
	35-40 M	37,620,063 1,416,214	22.1 0.3	3.0 1.2	287.5 118.9	44.3 19.7

reinforces our motivation for working with dense representations of factors and the corresponding tensor-based implementations.[13]

We generated Bayesian networks by starting with a linear order of nodes $V_1, \ldots, V_n$ and a maximum number of parents per node k. For each node V_i, we randomly determined a number of parents $\leq k$ and chose the parents randomly from the set $V_1, \ldots, V_{i-1}$. We then selected a fixed percentage f of non-root nodes and gave them functional CPTs, where each node had cardinality two or three.

In the first experiment, we considered networks with different number of nodes n, maximum number of parents k and percentage of functional nodes f. For each combination, we generated 10 networks, computed a binary jointree and averaged the size of largest cluster. We then applied our algorithm for exploiting functional dependencies and obtained a jointree with shrunk clusters and separators while also noting the size of largest cluster.

Table 5 depicts our results, where we report the size of a largest cluster in terms of its *binary rank:* log2 of its instantiations count. As can be seen from Table 5, our algorithm leads to substantial reductions in binary rank, where the reduction increases as the fraction of functional nodes increases. Recall that our algorithm includes two heuristics: one for deciding how to replicate functional CPTs when building a jointree and another corresponding to the choice on Line 14 in Figure 3. Table 5 provides some evidence on the efficacy of these heuristics beyond the rectangle and digits case studies we discussed earlier.

The second experiment compares classical and tensor-based implementations of ACs. In a classical implementation, the AC is represented as a directed acyclic graph where root nodes correspond to scalars and other nodes correspond to scalar arithmetic operations; see Figure 1. We will call this the *scalar graph* representation. In a tensor-based implementation, the AC is represented using a *tensor graph* where root nodes correspond to tensors (i.e., factors) and

[13] The experiments of this section were run on a server with dual Intel(R) Xeon E5-2670 CPUs running at 2.60GHz and 256GB RAM.

```
1: procedure EVALUATE_SCALAR_GRAPH(graph_nodes)
2:     for n in graph_nodes: do
3:         c1, c2 = n.child1, n.child2
4:         if n.type == 'add' then
5:             n.value = c1.value + c2.value
6:         else if n.type == 'mul' then
7:             n.value = c1.value * c2.value
8:         else if n.type == 'div' then
9:             n.value = c1.value / c2.value
10:        end if
11:    end for
12: end procedure
```

Figure 6: Evaluating a scalar graph representation of an AC. The graph nodes are topologically sorted so the children of a node are evaluated before the node is evaluated. The evaluation of a scalar-batch representation is similar except that node values are NumPy ndarrays and $+, *, /$ are ndarray (tensor) operations.

other nodes correspond to tensor operations (i.e., factor operations) as discussed in Section 4. The main benefit of a tensor-based implementation is that tensor operations can be parallelized on CPUs and GPUs (for example, NumPy and tools such as TensorFlow leverage Single Instruction Multiple Data (SIMD) parallelism on CPUs).[14]

Before we present the results of this experiment, we need to discuss the notion of a *batch* which is a set of AC input vectors. When learning the parameters of an AC using gradient descent, the dataset or a subset of it can be viewed as a batch so we would be interested in evaluating the AC on a batch. A scalar graph would need to be evaluated on each input vector in a batch separately. However, when representing the AC as a tensor graph we can treat the batch as a tensor. This allows us to evaluate the AC on a batch to yield a batch of marginals, which creates more opportunities for parallelism.

There is middle grounds though: a scalar graph with a batch that we shall call the *scalar-batch graph.* This is a tensor graph except that each tensor has two dimensions only: a batch dimension and a scalar dimension. For example, if the batch has size b, then a tensor will have shape $(b, 1)$. In a scalar-batch graph, each tensor is a set of scalars, one for each member of the batch (AC input vector).

Scalar-batch graphs can be used in situations where a full tensor graph cannot be used. This includes situations where the AC is compiled using techniques such as knowledge compilation, which produce ACs that cannot be cast in terms of tensor operations. A scalar-batch graph can be used in this case to offer an opportunity for parallelism, even if limited, especially when training the AC from data.

Table 6 compares the three discussed AC representations in terms of their evaluation time, while varying the batch size and AC size. The tensor graph implementation is the one we discussed in Section 4 using TensorFlow. The scalar graph implementation uses a Python list to store the DAG nodes (parents before children) and then uses the pseudocode in Figure 6 to evaluate the DAG. We extract the DAG from the tensor graph where each DAG node corresponds to a tensor entry. The scalar-batch graph is represented similarly to the scalar graph except that members of the list are NumPy ndarrays of shape $(b, 1)$ instead of scalars (we found NumPy to be more efficient than TensorFlow for this task). The evaluation time for both scalar graphs and scalar-batch graphs are therefore based on benchmarking the code in Figure 6 (we only measure the time of arithmetic operations, excluding setting evidence on root nodes and other overhead).

The networks in Table 6 were generated randomly as in the previous experiment, with 100 nodes and a maximum of 5 parents per node. For each limit on the AC size, we kept generating Bayesian networks randomly until we found 10 networks whose compilations yielded tensor graphs within the given size limit. The *tensor graph normalized time* in Table 6 is the total time to evaluate the graph divided by the batch size, then divided again by the size of the graph over $1000,000$. Normalized time is then the average time for evaluating one million AC nodes (tensor entries) and is meant to give a sense of speed independent of the batch and AC size.

We now have a number of observations on Table 6. The tensor graph is faster than the scalar and scalar-batch graphs in all cases and sometimes by two orders of magnitude. This can be seen in the last two columns of Table 6 which report the evaluation times (whole batch) of scalar and scalar-batch graphs over the evaluation time of tensor graph. The gap between tensor and scalar graphs increases with the batch size and with AC size as this means more opportunities to exploit parallelism on two fronts that the scalar graph cannot take advantage of. The gap between the tensor and scalar-batch graphs increases with AC size, but decreases with batch size. Increasing the AC size correlates with increasing the size of tensors (at least the largest one in the fourth column) which creates more opportunities for exploiting parallelism that the scalar-batch graph cannot exploit. However, increasing the batch size can be exploited by both the tensor and scalar-batch graphs, therefore narrowing the gap (NumPy appears to be exploiting the batch more effectively than TensorFlow). The scalar graph is faster than the scalar-batch graph when the batch size is 1, but otherwise is slower. This is to be expected as there is no need for the extra overhead of NumPy ndarrays in this case. We finally emphasize the absolute evaluation times for the tensor graph, which amount to a few milliseconds per one million AC nodes (normalized time) when the batch and AC size are large enough.

6 Conclusion

We presented new results on the algorithm of variable elimination that exploit functional dependencies using dense factors, allowing one to benefit from tensor-based technologies for more efficient inference and learning. We also presented case studies that show the promise of proposed techniques. In contrast to earlier approaches, the proposed one does not dependent on the identity of functional dependencies, only that they are present. This has further applications to exact inference (exploiting inferred functional dependencies) and to approximate inference (treating CPTs with extreme probabilities as functional CPTs) which we plan to pursue in future work.

ACKNOWLEDGEMENTS

I wish to thank members of the Automated Reasoning Group at UCLA who provided valuable motivation and feedback: Arthur Choi, Yizou Chen, Haiying Huang and Jason Shen. This work has been partially supported by grants from NSF IIS-1910317, ONR N00014-18-1-2561 and DARPA N66001-17-2-4032.

REFERENCES

[1] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle, 'Greedy layer-wise training of deep networks', in *Advances in Neural Information Processing Systems 19 (NIPS)*, pp. 153–160, (2006).

[2] Mark Chavira and Adnan Darwiche, 'On probabilistic inference by weighted model counting', *Artificial Intelligence*, **172**(6–7), 772–799, (April 2008).

[3] Mark Chavira, Adnan Darwiche, and Manfred Jaeger, 'Compiling relational bayesian networks for exact inference', *Int. J. Approx. Reasoning*, **42**(1-2), 4–20, (2006).

[14] https://en.wikipedia.org/wiki/SIMD

[4] Arthur Choi and Adnan Darwiche, 'On relaxing determinism in arithmetic circuits', in *Proceedings of the Thirty-Fourth International Conference on Machine Learning (ICML)*, pp. 825–833, (2017).
[5] Arthur Choi and Adnan Darwiche, 'On the relative expressiveness of bayesian and neural networks', in *PGM*, volume 72 of *Proceedings of Machine Learning Research*, pp. 157–168. PMLR, (2018).
[6] Arthur Choi, Ruocheng Wang, and Adnan Darwiche, 'On the relative expressiveness of bayesian and neural networks', *International Journal of Approximate Reasoning*, **113**, 303–323, (2019).
[7] Adnan Darwiche, 'Recursive conditioning', *Artif. Intell.*, **126**(1-2), 5–41, (2001).
[8] Adnan Darwiche, 'A logical approach to factoring belief networks', in *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR)*, pp. 409–420, (2002).
[9] Adnan Darwiche, 'A differential approach to inference in Bayesian networks', *Journal of the ACM (JACM)*, **50**(3), 280–305, (2003).
[10] Adnan Darwiche, *Modeling and Reasoning with Bayesian Networks*, Cambridge University Press, 2009.
[11] Rina Dechter, 'Bucket elimination: A unifying framework for probabilistic inference', in *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 211–219, (1996).
[12] Rina Dechter, *Constraint processing*, Elsevier Morgan Kaufmann, 2003.
[13] Rina Dechter and Yousri El Fattah, 'Topological parameters for time-space tradeoff', *Artif. Intell.*, **125**(1-2), 93–118, (2001).
[14] Jeffrey M. Dudek, Leonardo Dueñas-Osorio, and Moshe Y. Vardi, 'Efficient contraction of large tensor networks for weighted model counting through graph decompositions', *CoRR*, **abs/1908.04381**, (2019).
[15] Glen Evenbly and Robert N. C. Pfeifer, 'Improving the efficiency of variational tensor network algorithms', *Phys. Rev. B*, **89**, 245118, (Jun 2014).
[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.
[17] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh, 'A fast learning algorithm for deep belief nets', *Neural Computation*, **18**(7), 1527–1554, (2006).
[18] F. V. Jensen, S. Lauritzen, and K. Olesen, 'Bayesian updating in recursive graphical models by local computation', *Computational Statistics Quarterly*, **4**, 269–282, (1990).
[19] Frank Jensen and S. Anderson, 'Approximations in bayesian belief universe for knowledge based systems', in *Proceedings of the Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-90)*, pp. 162–169, Corvallis, Oregon, (1990). AUAI Press.
[20] Uffe Kjærulff, 'Triangulation of graphs – algorithms giving small total state space', Technical report, (1990).
[21] David Larkin and Rina Dechter, 'Bayesian inference in the presence of determinism', in *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, (2003).
[22] Vasilica Lepar and Prakash P. Shenoy, 'A comparison of lauritzen-spiegelhalter, hugin, and shenoy-shafer architectures for computing marginals of probability distributions', in *UAI*, pp. 328–337. Morgan Kaufmann, (1998).
[23] Judea Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, MK, 1988.
[24] Judea Pearl, *Causality*, Cambridge University Press, 2000.
[25] Judea Pearl and Dana Mackenzie, *The Book of Why: The New Science of Cause and Effect*, Basic Books, 2018.
[26] Marc'Aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, and Yann LeCun, 'Efficient learning of sparse representations with an energy-based model', in *Advances in Neural Information Processing Systems 19 (NIPS)*, pp. 1137–1144, (2006).
[27] Yujia Shen, Haiying Huang, Arthur Choi, and Adnan Darwiche, 'Conditional independence in testing bayesian networks', in *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pp. 5701–5709. PMLR, (2019).
[28] Prakash P. Shenoy, 'Binary join trees', in *UAI*, pp. 492–499. Morgan Kaufmann, (1996).
[29] Nevin Lianwen Zhang and David Poole, 'Exploiting causal independence in bayesian network inference', *Journal of Artificial Intelligence Research*, **5**, 301–328, (1996).