A Logical Approach to Factoring Belief Networks

Adnan Darwiche
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
darwiche@cs.ucla.edu

Abstract

We have recently proposed a tractable logical form, known as deterministic, decomposable negation normal form (d-DNNF). We have shown that d-DNNF supports a number of logical operations in polynomial time, including clausal entailment, model counting, model enumeration, model minimization, and probabilistic equivalence testing. In this paper, we discuss another major application of this logical form: the implementation of multi-linear functions (of exponential size) using arithmetic circuits (that are not necessarily exponential). Specifically, we show that each multi–linear function can be encoded using a propositional theory, and that each d-DNNF of the theory corresponds to an arithmetic circuit that implements the encoded multi–linear function. We discuss the application of these results to factoring belief networks, which can be viewed as multi–linear functions as has been shown recently. We discuss the merits of the proposed approach for factoring belief networks, and present experimental results showing how it can handle efficiently belief networks that are intractable to structure–based methods for probabilistic inference.

1 Introduction

We have recently proposed a tractable logical form, known as deterministic, decomposable negation normal form (d-DNNF) [9, 13]. We have shown that d-DNNF supports a number of logical operations in polynomial time, including clausal entailment, model counting, model enumeration, model minimization, and probabilistic equivalence testing [9, 13, 12]. In this paper, we discuss another major application of this logical form: the implementation of multi-linear functions of exponential size in terms of arithmetic circuits that are not necessarily exponential. A multi–linear function is multi–variate polynomial in which each variable has degree one—that is, the polynomial is linear in each of its variables. An arithmetic circuit is a rooted, directed acyclic graph in which leaf nodes correspond to circuit inputs, internal nodes correspond to arithmetic operations, and the root corresponds to the circuit output. One can implement a multi–linear function using an arithmetic circuit. This implementation is interesting because a multi–linear function that has an exponential size may be implemented by an arithmetic circuit whose size is not necessarily exponential. The relationship between polynomials and arithmetic circuits that implement them is a classical subject of algebraic complexity theory [30], where one is interested in the circuit complexity of certain problems—that is, the size of the smallest arithmetic circuits that can solve certain problems. Here, we will focus on the problem of generating an arithmetic circuit that implements a given multi–linear function.

The central result in this paper has two parts. First, we show that each multi–linear function can be encoded using a propositional theory. Second, we show that each d-DNNF of the theory corresponds to an arithmetic circuit which implements the encoded multi–linear function. Hence, an algorithm that converts a propositional theory into d-DNNF is immediately an algorithm for generating arithmetic circuit implementations of multi–linear functions.

We prove this result first and then discuss one of its major applications to probabilistic inference. Specifically, we have shown recently that a belief network can be represented algebraically as a multi–linear function [7, 6]. We have also shown that a large number of probabilistic queries can be obtained immediately from the partial derivatives of such a function.

The problem however is that the multi–linear function has an exponential size, which makes it mostly of semantical interest. But if we can implement this function efficiently using an arithmetic circuit, the approach then becomes of main computational interest since the partial derivatives of an arithmetic circuit can all be computed simultaneously in time linear in the circuit size [27]. Our central result in this regard is that the multi–linear function of a belief network can be encoded efficiently using a propositional theory in Conjunctive Normal Form (CNF). Hence, by converting such a CNF into a d-DNNF of small size, we are able to compile the belief network into an arithmetic circuit, on which we can perform inference in linear time. As it turns out, the proposed approach allows us to efficiently compile some belief networks that are intractable to structure–based inference algorithms for belief networks.

This paper is structured as follows. We start by discussing deterministic, decomposable negation normal form in Section 2. We then show how multi–linear functions can be encoded using propositional theories in Section 3, and how the d-DNNF of such a theory corresponds to an arithmetic circuit implementation of the function it encodes. Section 4 is then dedicated to the application of these results to belief network inference. Experimental results are given in Section 5. We finally close with some concluding remarks in Section 6. Proofs of theorems are delegated to Appendix A.

2 Deterministic, decomposable negation normal form

We review in this section the logical form known as deterministic, decomposable negation normal form (d-DNNF) [9, 13], and discuss its relationship to the well-known Binary Decision Diagram (BDD) [4].

A negation normal form (NNF) is a rooted, directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction $\wedge$ or disjunction $\vee$. Figure 1 depicts an example. For any node n in an NNF graph, $\mathit{Vars}(n)$ denotes all propositional variables that appear in the subgraph rooted at n, and $\Delta(n)$ denotes the formula represented by n and its descendants. Consider the marked node n in Figure 1(b). We have $\mathit{Vars}(n) = A, B$ and $\Delta(n) = (\neg A \wedge B) \vee (A \wedge \neg B)$. A number of properties can be stated on NNF graphs:

- Decomposability holds when $\mathit{Vars}(n_i) \cap \mathit{Vars}(n_j) = \emptyset$ for any two children n_i and n_j of an and-node n. This can be verified for the children of marked node in Figure 1(a).
- Determinism holds when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j of an or-node n. This can be verified for the children of marked node in Figure 1(b).
- Smoothness holds when $\mathit{Vars}(n_i) = \mathit{Vars}(n_j)$ for any two children n_i and n_j of an or-node n. This can be verified for the children of marked node in Figure 1(c).
- Decision holds when the root node of the NNF graph is a decision node. A *decision node* is a node labeled with *true*, *false*, or is an or-node having the form or(and(X, α), and(¬X, β)), where X is a variable, α and β are decision nodes. Here, X is called the *decision variable* of the node.
- Ordering is defined only for NNFs that satisfy the decision property. Ordering holds when decision variables appear in the same order along any path from the root to any leaf.

The class of decomposable negation normal form (DNNF) has been identified and studied in [8]. The class of deterministic, decomposable negation normal form (d-DNNF) has been identified and studied in [9]. Decision implies determinism. The subset of NNF that satisfies decomposability and decision (hence, determinism) corresponds to Free Binary Decision Diagrams (FBDDs) [15, 13]. The subset of NNF that satisfies decomposability, decision (hence, determinism) and ordering corresponds to Ordered Binary Decision Diagrams (OBDDs) [4, 13]. In BDD notation, however, the NNF fragment or(and(X, α), and(¬X, β)) is drawn more compactly as X(α, β). Hence, each internal BDD node generates three NNF nodes and six NNF edges.

Immediate from the above definitions, we have the following strict subset inclusions OBDD $\subset$ FBDD $\subset$ d-DNNF $\subset$ DNNF. Moreover, DNNF $\succ$ d-DNNF $\succ$ FBDD $\succ$ OBDD, where $\succ$ stands for "more succinct than."[1] Language L_1 is more succinct than another L_2, $L_1 \succ L_2$, iff any sentence α in L_2 has an equivalent sentence β in L_1 whose size is polynomial in the size of α; and the opposite is not true [13]. The smoothness property is not critical computationally, as it can be enforced in polynomial time [9], yet is quite helpful as we see later. The reader is referred to [13] for a comprehensive analysis of these forms and their relationships.

[1] That DNNF is strictly more succinct than d-DNNF assumes the non-collapse of the polynomial heirarchy [13].

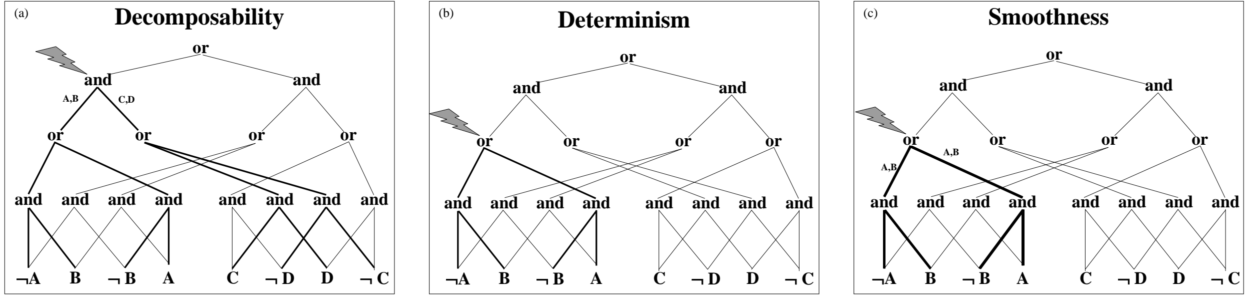


Figure 1: A negation normal form graph that satisfies decomposability, determinism and smoothness.

An algorithm for converting propositional theories in Conjunctive Normal Form (CNF) into deterministic, decomposable negation normal form (d-DNNF) is given in [10], and is used in the experimental results reported in Section 5.

3 Multi–linear functions and arithmetic circuits

We have two main goals in this section. Our first goal is to formally introduce multi–linear functions and their implementation using arithmetic circuits. Our second goal is to show how d-DNNF can be used to realize this implementation process.

A *multi–linear function* over variables Σ is a function of the form $t_1 + t_2 + \dots t_n$, where each *term* t_i is a product of distinct variables from Σ. For example, if $\Sigma = a, b, c$, then $a+bc+ac+abc$ is a multi–linear function. There are $2^{|\Sigma|}$ distinct terms and $2^{2^{|\Sigma|}}$ distinct multi–linear functions over variables Σ.[2] Some of the multi–linear functions that come up in practice have an exponential number of terms. For example, it is shown in [7] that a belief network can be interpreted as a multi–linear function that has an exponential number of terms.

To reason with such multi–linear functions, one needs a more efficient representation. The arithmetic circuit is such a representation. An *arithmetic circuit* is a rooted, directed acyclic graph where leaves are labelled with numbers or variables, and non-leaves are labelled with arithmetic operations; see Figure 2 (right). We restrict our attention here to only addition and multiplication operations. An arithmetic circuit *implements* a multi–linear function f over variables Σ if for every value σ of these variables, the circuit will output $f(\sigma)$ under input σ. The circuit in Figure 2 implements the multi–linear function $ac + abc + c$. In the terminology of algebraic complexity theory [30], one says that the circuit in Figure 2 *computes* the function $ac + abc + c$.

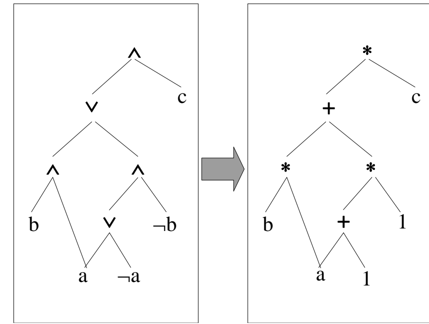


Figure 2: Extracting an arithmetic circuit from a smooth d-DNNF.

We now turn to the process of generating an arithmetic circuit representation of a given multi–linear function. Our starting point is to show that every propositional theory over variables Σ can be interpreted as encoding some multi–linear function over the same variables Σ. Consider the theory $\Delta = (a \vee \neg b) \wedge c$ over variables $\Sigma = a, b, c$ as an example. This theory has three models:

- $\sigma_1 : a = true, b = false, c = true$;
- $\sigma_2 : a = true, b = true, c = true$; and
- $\sigma_3 : a = false, b = false, c = true$.

Each one of these models σ_i can be interpreted as encoding a term t_i in the following sense. A variable appears in the term t_i iff the model σ_i sets the variable to true. That is, the first model encodes the term ac. The second model encodes the term abc, and the third model encodes the term c. The theory Δ then encodes the multi–linear function that results from adding up all these terms: $ac + abc + c$.

[2]Without loss of generality, we disallow duplicate terms in the representation of multi–linear functions.

Definition 1 *Let σ be a truth assignment over variables Σ. The* <u>*term*</u> *encoded by σ is defined as follows:*

$$t(\sigma) \stackrel{def}{=} \prod_{\sigma(V)=true} V.$$

Let Δ be a propositional theory over variables Σ. The <u>*multi–linear function*</u> *encoded by Δ is defined as follows:*

$$f(\Delta) \stackrel{def}{=} \sum_{\sigma \models \Delta} t(\sigma).$$

Propositional theories can then be used as a *specification* language for multi–linear functions. In particular, it is possible to encode a multi–linear function that has an exponential number of terms using a compact propositional theory (which has an exponential number of models). We show in Section 4 for example that the multi–linear functions corresponding to belief networks, which have an exponential number of terms, can be encoded using CNF theories of linear size.

We now have the following central result.

Definition 2 *Let Δ be a smooth d-DNNF. The* <u>*arithmetic circuit*</u> *encoded by Δ is obtained by replacing each conjunction in Δ by multiplication, each disjunction by addition, and each negative literal by 1.*

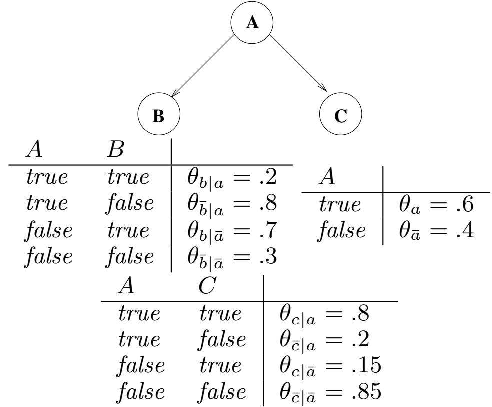
Figure 2 contains a smooth d-DNNF (left) and the arithmetic circuit it encodes (right).

Theorem 1 *Let Δ be a smooth d-DNNF which encodes a multi-linear function f. The arithmetic circuit encoded by Δ implements the function f.*

Therefore, if we have a multi–linear function f which is specified/encoded by a propositional theory Γ (in any form), we can generate a circuit implementation of this function by converting Γ into smooth d-DNNF. Moreover, the generated circuit has the same size as the d-DNNF. Hence, by optimizing the size of d-DNNF we are also optimizing the size of generated circuit. As mentioned earlier, we have presented a CNF to d-DNNF compiler in [10] which is quite effective on a variety of propositional theories. We use this compiler in Section 5 to compile belief networks into arithmetic circuits.

4 Factoring belief networks

We have three goals in this section. Our first goal is to review results from [7], which show that a belief network can be interpreted as a multi–linear function, and that a large number of probabilistic queries can be computed immediately using the partial derivatives of this multi–linear function. The second goal is to show that the multi–linear function of a belief network can be encoded efficiently using a CNF. Hence, a CNF to d-DNNF compiler can then be used immediately for compiling a belief network into an arithmetic circuit, which can then be used for linear time inference. Our third goal is to discuss the merits of presented approach to inference in belief networks, as compared to structure–based inference algorithms for this purpose.



A	B	
true	*true*	$\theta_{b\mid a} = .2$
true	*false*	$\theta_{\bar{b}\mid a} = .8$
false	*true*	$\theta_{b\mid \bar{a}} = .7$
false	*false*	$\theta_{\bar{b}\mid \bar{a}} = .3$

A	
true	$\theta_a = .6$
false	$\theta_{\bar{a}} = .4$

A	C	
true	*true*	$\theta_{c\mid a} = .8$
true	*false*	$\theta_{\bar{c}\mid a} = .2$
false	*true*	$\theta_{c\mid \bar{a}} = .15$
false	*false*	$\theta_{\bar{c}\mid \bar{a}} = .85$

Figure 3: A belief network with its CPTs.

4.1 Belief networks as multi–linear functions

A belief network is a factored representation of a probability distribution [24]. It consists of two parts: a directed acyclic graph (DAG) and a set of conditional probability tables (CPTs). For each node X and its parents $\mathbf{U}$ in the DAG, we must have a CPT that specifies the probability distribution of X under each instantiation $\mathbf{u}$ of the parents.[3] Figure 3 depicts a simple belief network which has three CPTs.

A belief network is a "representational" factorization of a probability distribution, not a "computational" one. That is, although the network allows us to compactly represent the distribution, it needs to be processed further if one is to obtain answers to arbitrary probabilistic queries. Mainstream algorithms for inference in belief networks operate on the network to generate a "computational" factorization, allowing one to answer queries in time which is linear in the factor-

[3]We are using the standard notation: variables are denoted by upper–case letters (A) and their values by lower–case letters (a). Sets of variables are denoted by bold–face upper–case letters ($\mathbf{A}$) and their instantiations are denoted by bold–face lower–case letters ($\mathbf{a}$). For a variable A with values *true* and *false*, we use a to denote A= *true* and $\bar{a}$ to denote A= *false*. Finally, for a variable X and its parents $\mathbf{U}$, we use $\theta_{x\mid\mathbf{u}}$ to denote the CPT entry corresponding to $Pr(x \mid \mathbf{u})$.

ization size. One of the most influential computational factorizations of a belief network is the *jointree* [18, 17]. Standard jointree factorizations are structure-based; that is, their size depend only on the network topology and is invariant to local CPT structure. This observation has triggered much research recently for alternative, finer–grained factorizations, since real-world networks can exhibit significant local structure that can lead to significant computational savings [21, 5, 26, 31].

We discuss next one of the latest proposals in this direction, which calls for using arithmetic circuits as a computational factorization of belief networks [7, 6]. This proposal is based on viewing each belief network as a multi-linear function, which can be implemented using an arithmetic circuit. The function itself contains two types of variables:

- *Evidence indicators:* For each variable X in the network , we have a variable λ_x for each value x of X.
- *Network parameters:* For each variable X and its parents $\mathbf{U}$ in the network, we have a variable $\theta_{x|\mathbf{u}}$ for each value x of X and instantiation $\mathbf{u}$ of $\mathbf{U}$.

The multi–linear function has a term for each instantiation of the network variables, which is constructed by multiplying all evidence indicators and network parameters that are consistent with that instantiation. For example, the multi–linear function of the network in Figure 3 has 8 terms corresponding to the 8 instantiations of variables A, B, C. Three of these terms are shown below:

$$\begin{aligned} f &= \lambda_a \lambda_b \lambda_c \theta_a \theta_{b|a} \theta_{c|a} + \\ & \quad \lambda_a \lambda_b \lambda_{\bar{c}} \theta_a \theta_{b|a} \theta_{\bar{c}|a} + \\ & \quad \ldots \\ & \quad \lambda_{\bar{a}} \lambda_{\bar{b}} \lambda_{\bar{c}} \theta_{\bar{a}} \theta_{\bar{b}|\bar{a}} \theta_{\bar{c}|\bar{a}}. \end{aligned}$$

Given this multi–linear function f, we can answer any query with respect to its corresponding belief network. Specifically, let $\mathbf{e}$ be an instantiation of some network variables, and suppose we want to compute the probability of $\mathbf{e}$. We can do this by simply evaluating the multi–linear function f while setting each evidence indicator λ_x to 1 if x is consistent with $\mathbf{e}$, and to 0 otherwise. For the network in Figure 3, we can compute the probability of evidence $\mathbf{e} = b\bar{c}$ by evaluating its multi–linear function above under $\lambda_a = 1, \lambda_{\bar{a}} = 1, \lambda_b = 1, \lambda_{\bar{b}} = 0$ and $\lambda_c = 0, \lambda_{\bar{c}} = 1$. This leads to $f = \theta_a \theta_{b|a} \theta_{\bar{c}|a} + \theta_{\bar{a}} \theta_{b|\bar{a}} \theta_{\bar{c}|\bar{a}}$, which equals the probability of $b, \bar{c}$ in this case. We use $f(\mathbf{e})$ to denote the result of evaluating the function f under evidence $\mathbf{e}$ as given above.

This algebraic representation of belief networks is attractive as it allows us to obtain answers to a large number of probabilistic queries directly from the derivatives of the multi–linear function. For example, the probability of any instantiation $\mathbf{e}, x$, where $X \notin \mathbf{E}$, is nothing but the partial derivative $\partial f / \partial \lambda_x$ evaluated at $\mathbf{e}$, denoted $\partial f / \partial \lambda_x(\mathbf{e})$. Moreover, the probability of any instantiation $\mathbf{e}, x, \mathbf{u}$, where $\mathbf{U}$ are the parents of X, is nothing but $\theta_{x|\mathbf{u}} \partial f / \partial \theta_{x|\mathbf{u}}(\mathbf{e})$. The ability to compute answers to probabilistic queries directly from such derivatives is valuable computationally since the (first) partial derivatives of an arithmetic circuit can all be computed simultaneously in time linear in the circuit size [7, 27]. Therefore, by implementing the multi–linear function as an arithmetic circuit, we can compute a large number of probabilistic queries in time linear the circuit size.

4.2 Encoding a belief network using a CNF

The multi-linear function of a belief network has an exponential number of terms. Yet, one can encode it efficiently using a CNF. The encoding is given next.

Definition 3 *The CNF encoding Δ of a belief network contains two types of propositional variables. For each network variable X, we have a propositional variable λ_x for each value x of X. And for each network variable X and its parents $\mathbf{U}$, we have a propositional variable $\theta_{x|\mathbf{u}}$ for each value x of X and instantiation $\mathbf{u}$ of $\mathbf{U}$. For each network variable X with values $x^1, \ldots, x^k$, the encoding Δ contains the clauses:*

$$\lambda_{x^1} \vee \ldots \vee \lambda_{x^k} \tag{1}$$

$$\neg\lambda_{x^i} \vee \neg\lambda_{x^j}, \quad i \neq j. \tag{2}$$

Moreover, for each propositional variable $\theta_{x|u_1,\ldots,u_m}$, the encoding Δ contains the clauses:

$$\lambda_x \wedge \lambda_{u_1} \wedge \ldots \wedge \lambda_{u_m} \rightarrow \theta_{x|u_1,\ldots,u_m} \tag{3}$$

$$\theta_{x|u_1,\ldots,u_m} \rightarrow \lambda_x \tag{4}$$

$$\theta_{x|u_1,\ldots,u_m} \rightarrow \lambda_{u_1}, \quad \ldots, \quad \theta_{x|u_1,\ldots,u_m} \rightarrow \lambda_{u_m}. \tag{5}$$

Clauses 1–2 say that each term of the multi–linear function must include exactly one of the evidence indicators for X. Clauses 3–5 say that a term includes evidence indicators λ_x, λ_{u_1}, $\ldots \lambda_{u_m}$ iff it includes parameter $\theta_{x|u_1,\ldots,u_m}$. Clauses 3–5 can be compactly represented as an equivalence:

$$\lambda_x \wedge \lambda_{u_1} \wedge \ldots \wedge \lambda_{u_m} \leftrightarrow \theta_{x|u_1,\ldots,u_m}. \tag{6}$$

Theorem 2 *The CNF encoding of a belief network encodes the multi–linear function of given network.*

Given that each network variable has at most k values and at most m parents, the CNF encoding contains $O(nmk^m + nk^2)$ clauses. $O(nk^2)$ of these clauses are of Types 1–2, and $O(nmk^m)$ clauses are of Types 3–5. Note that the number of CPT entries is $O(nk^m)$ in this case. The encoding suggested by Definition 3 is practical when we do not have many values (k) or parents (m) per variable. When either k or m is not small enough, a much more efficient encoding would result if one uses a constraint–based language which allows multi-valued variables. We do not purse this approach here though, as it would require a d-DNNF compiler which can handle multi–valued variables. The d-DNNF compiler we used from [10] does not handle such variables yet.

Let us now consider an example encoding for a two–node network $A \rightarrow B$, where A has values $a, \bar{a}$ and B has values $b, \bar{b}$. The CNF encoding Δ has the following clauses:

$$\begin{aligned} \lambda_a \vee \lambda_{\bar{a}}, \quad \neg\lambda_a \vee \neg\lambda_{\bar{a}} \\ \lambda_b \vee \lambda_{\bar{b}}, \quad \neg\lambda_b \vee \neg\lambda_{\bar{b}} \\ \lambda_a \leftrightarrow \theta_a \\ \lambda_{\bar{a}} \leftrightarrow \theta_{\bar{a}} \\ \lambda_a \wedge \lambda_b \leftrightarrow \theta_{b|a} \\ \lambda_a \wedge \lambda_{\bar{b}} \leftrightarrow \theta_{\bar{b}|a} \\ \lambda_{\bar{a}} \wedge \lambda_b \leftrightarrow \theta_{b|\bar{a}} \\ \lambda_{\bar{a}} \wedge \lambda_{\bar{b}} \leftrightarrow \theta_{\bar{b}|\bar{a}}. \end{aligned} \tag{7}$$

This CNF has four models and encodes the multi-linear function:

$$f = \lambda_a\lambda_b\theta_a\theta_{b|a} + \lambda_a\lambda_{\bar{b}}\theta_a\theta_{\bar{b}|a} + \lambda_{\bar{a}}\lambda_b\theta_{\bar{a}}\theta_{b|\bar{a}} + \lambda_{\bar{a}}\lambda_{\bar{b}}\theta_{\bar{a}}\theta_{\bar{b}|\bar{a}}.$$

Note that a variable such as $\theta_{b|\bar{a}}$ is interpreted as a propositional variable when it appears in the CNF Δ, but is interpreted as a real variable with values in $[0, 1]$ when it appears in the multi–linear function f.

The encoding given in Definition 3 depends only on the network structure and the domains of its variables. Hence, two networks with the same structure and domains will generate the same encoding. We show next how to refine this encoding in order to exploit non–structural network properties: logical constraints and context–specific independence.

Logical constraints. A logical constraint corresponds to a conditional probability which is equal to either 0 or 1. One can produce a more efficient CNF encoding in the presence of logical constraints. Suppose for example that a network parameter $\theta_{x|u_1,\ldots,u_m}$ takes the value 0. We can then replace Clauses 3–5 by a single clause:

$$\neg\lambda_x \vee \neg\lambda_{u_1} \vee \ldots \vee \neg\lambda_{u_m}.$$

Hence, the variable $\theta_{x|u_1,\ldots,u_m}$ is eliminated from the encoding and the number of clauses is reduced. We can do this since every term which includes the indicators $\lambda_x, \lambda_{u_1}, \ldots, \lambda_{u_m}$ is multiplied by 0. Since these terms do not contribute to the multi–linear function value, there is no harm in excluding them. The above clause achieves this effect by excluding models that encode these terms. Similarly, if $\theta_{x|u_1,\ldots,u_m} = 1$, we can omit the parameter $\theta_{x|u_1,\ldots,u_m}$ and its associated clauses 3–5 from the encoding. The clauses only say that every term which includes the indicators $\lambda_x, \lambda_{u_1}, \ldots, \lambda_{u_m}$ is multiplied by $1 = \theta_{x|u_1,\ldots,u_m}$. It is then safe to eliminate this parameter and the corresponding clauses as they are vacuous in this case. As we show in Section 5, not only do logical constraints lead to a more efficient encoding, but they also lead to a major reduction in the size of compiled d-DNNF and its corresponding arithmetic circuit. This is a major advantage of our proposed approach for probabilistic inference as it can exploit logical constraints computationally, without requiring any algorithmic refinements. The exploitation takes place at the encoding level.

Context–specific independence. Suppose we have a variable X with two parents Y and Z in a belief network. Suppose further that $\theta_{x|yz} = \theta_{x|y\bar{z}}$. This means that given y, our belief in x is independent of Z. This form of independence is known as *context–specific independence (CSI)* [3] and is quite different from *structural independence* since we cannot identify it by examining the network structure. There is evidence that CSI is very valuable computationally [5, 31], yet it has proven hard to exploit it in the context of pure structure–based algorithms, such as the jointree algorithm [18, 17]. One can exploit CSI quite easily in the proposed framework. Specifically, all we have to do in the previous case is to replace the parameters $\theta_{x|yz}$ and $\theta_{x|y\bar{z}}$ in the CNF encoding by a new parameter, $\theta_{x|y}$. Moreover, we replace the clauses for these two parameters with $\lambda_x \wedge \lambda_y \leftrightarrow \theta_{x|y}$. We show in Section 5 the dramatic effect that CSI has on the size of d-DNNF and its corresponding arithmetic circuit.

We close this section by stressing that the CNF encoding of a belief network can be generated automatically once we know the network structure and the domain of each network variable. To incorporate logical constraints, however, we also need to know whether a network parameter attains the value 0 or 1. And to incorporate context–specific independence, we need to known which network parameters are equal.

4.3 A new approach to probabilistic inference

We now summarize our proposed approach for inference in belief networks. Given a belief network N, our

approach amounts to the following steps:

1. Encode the multi–linear function of N using a propositional theory Δ as given in Definition 3. Again, this can be completely automated with no need for human intervention.
2. Compile the encoding Δ into a smooth d-DNNF Γ using a compiler such as the one in [10]. One can use an OBDD package for this purpose too since OBDDs are a special case of d-DNNFs.[4] OBDDs are less succinct though, leading to arithmetic circuits which are larger than is really needed.[5]
3. Extract an arithmetic circuit Υ from the smooth d-DNNF Γ as given in Definition 2.
4. Use the circuit Υ for linear-time inference as described in [7, 6].

We present experimental results in the following section, which illustrate the effectiveness of our proposed approach.

We close this section by mentioning two other approaches for generating circuit implementations of the multi–linear functions corresponding to belief networks. First, we have shown in [7] that if we have a belief network with n variables, and an elimination order of width w for the network, then we can generate an arithmetic circuit that implements the network multi–linear function in $O(n \exp(w))$ time and space.[6] Second, we have shown recently that a carefully constructed jointree and a choice of one of its clusters can be viewed as an implicit circuit implementation of the network multi–linear function (where separator entries correspond to addition nodes and cluster entries correspond to multiplication nodes) [23]. Moreover, we have shown that a message-collect towards the chosen cluster corresponds to a circuit evaluation, while a message-distribute from the cluster corresponds to a circuit differentiation. Hence, the jointree method can be viewed as a special case of the framework proposed here, where the construction of a jointree corresponds to a specific method for constructing circuit implementations of the network multi–linear function. The arithmetic circuits generated by jointrees are interesting as they have a "regular" structure, allowing one to represent them without having to explicitly represent their edges (edges can be induced from the jointree structure). Using jointrees however (and any other structure–based method) for generating circuits is much less attractive in the presence of logical constraints and context–specific independence. More on this point in the following section.

[4]We also need to smooth the OBDD, which can be done in polynomial time [9].

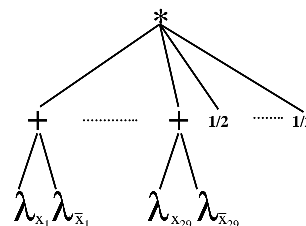
[5]The arithmetic circuit extracted from an OBDD is closely related to an algebraic decision diagram (ADD) [2]. See also [22, 16] for proposals on using OBDDs and ADDs as part of algorithms for probabilistic inference.

[6]One way to define the *width* of an elimination order is as the size of the maximum clique -1 in the jointree constructed based on the order.

5 Experimental results

We have two main goals in this section. Our first goal is to highlight the difference between our proposed approach for inference in belief networks, and structure–based approaches, such as those based on jointrees. In particular, we will present a few specific examples which illustrate the extent to which the proposed approach is sensitive to non–structural properties of belief networks: logical constraints and context–specific independence. Our second goal in this section is to provide statistics on the arithmetic circuits we were able to build for belief networks that are completely outside the scope of structure–based approaches. These networks correspond to digital circuits, with jointrees that have clique sizes > 60 (the complexity of the jointree algorithm is exponential in the size of maximum clique size).

We start with an example that we found to be quite revealing: the *printer_ts* belief network, distributed with the evaluation version of HUGIN (*www.hugin.com*). The network has 29 binary variables, call them $X_1, \ldots, X_{29}$, and 272 parameters. HUGIN compiles the network into a jointree with 11 cliques and 10 separators. The total size of clique tables is 236, and the total size of separator tables is 20. This means that if we were to construct an arithmetic circuit based on this jointree, as described in [23], the circuit will have 236 multiplication nodes and 20 addition nodes. The circuit we obtained using our approach had only 1 multiplication node and 29 addition nodes:



This appeared surprising at first, but it turns out that all parameters in the *printer_ts* network are actually set to 1/2, which means that the network is full of context–specific independence (this is probably not intended). Hence, even though the multi–linear function

of the network has 2^{29} terms, the function has a simple factorization: $(1/2)^{29} \prod_{i=1}^{29}(\lambda_{x_i} + \lambda_{\bar{x}_i})$. In fact, we would obtain this factorization, which size is linear in the number of network variables, regardless of the belief network structure. This example shows that our approach can generate a circuit of linear size for a belief network of arbitrary structure, if enough context–specific independence exists in the network.

Our second set of examples concern the extent to which logical constraints in a belief network affect the size of its compiled arithmetic circuit. Here we consider a few networks which are distributed with the evaluation version of HUGIN: *poker*, *golf*, *boblo*, *6nt*, *6hj* and *3nt*. Each one of these networks has a large number of 0/1 parameters. For example, the network *golf* has 8 multi–valued nodes, three of which have CPTs with only 0/1 parameters. We show below the size of arithmetic circuit we obtained for each network (number of addition/multiplication nodes), using the logic–based method proposed in this paper, in addition to the structure–based method based on jointrees [23]. The jointrees were computed by Netica (*www.norsys.com*).

Net	Vars#	d-DNNF–based $\star$/+nodes	Jointree–based $\star$/+nodes
Poker	7	302	685
Golf	8	143	676
Boblo	22	393	494
6nt	58	1377	12378
6hj	58	1814	29176
3nt	58	6328	35902

As is clear from the above table, logical constraints lead to significant reductions in the size of arithmetic circuits, and our proposed method appears to naturally exploit these constraints.

The computational value of logical constraints to probabilistic reasoning has long been observed. In fact, even structure–based methods have been refined to exploit such constraints. For example, a method known as *zero compression* has been proposed for jointrees [19], which reduces the size of a jointree in the presence of logical constraints.

We now consider a number of real–world belief networks which correspond to digital circuits that have very high connectivity. The goal here is to reason about the probability that any particular wire will have a certain signal on it, given evidence about other wires in the circuit. All we are given, in addition to the circuit, is a probability distribution over each circuit input. A belief network for this reasoning application is constructed in a systematic way, by including a network variable for each wire in the circuit and then adding an edge from each gate's input to its output. The CPT entries for each root node X in the network—that is, θ_x and $\theta_{\bar{x}}$—are determined based on the given distribution for the corresponding circuit input. All other CPT entries are either 0 or 1, and are determined based on the gate corresponding to that CPT. Belief networks which have 0/1 parameters everywhere, except possibly for the parameters of root variables, are known as *deterministic belief networks.* Such networks appear extensively in applications relating to causal and diagnostic reasoning [25]. A belief network where every node is related to its parents by a *noisy–or* model can also be easily reduced to a deterministic belief network [24].

We can build arithmetic circuits for deterministic belief networks as described in the previous section, but there is a much more efficient encoding method which we describe and use next.

Definition 4 *Let N be a deterministic belief network with variables $X_1, \ldots, X_n$, where each variable has two values: positive and negative. The CNF encoding of network N is a propositional theory over variables $X_1, \ldots, X_n$, which includes the clause $l(x) \vee l(u_1) \vee \ldots l(u_m)$ iff the parameter $\theta_{x|u_1,\ldots,u_m}$ equals 0. Here, $l(x) = \neg X$, when x is the positive value of variable X, and $l(x) = X$, when x is the negative value of X.*

This encoding is much more efficient than the one given in Definition 3 as it only includes one propositional variable for every network variable. It also includes a smaller number of clauses. Extracting an arithmetic circuit from this encoding is also a bit different from what is given in Definition 2.

Definition 5 *Let Δ be a smooth d-DNNF which encodes a deterministic belief network N over variables $X_1, \ldots, X_n$. Let x_i denote the positive value of X_i and $\bar{x}_i$ denote its negative value. The arithmetic circuit encoded by Δ is obtained by replacing every conjunction with a multiplication; every disjunction with an addition; every leaf node X_i with $\lambda_{x_i} \star \theta_{x_i}$ if X_i is a root in N, and with λ_{x_i} otherwise; every leaf node $\neg X$ with $\lambda_{\bar{x}_i} \star \theta_{\bar{x}_i}$ if X is a root in N, and with $\lambda_{\bar{x}_i}$ otherwise.*

Theorem 3 *The arithmetic circuit described in Definition 5 implements the multi–linear function of the corresponding deterministic belief network.*

Consider a circuit which consists of only one exclusive–or gate, leading to the structure $X_1 \rightarrow X_3 \leftarrow X_2$. The CNF encoding of this network will be as follows:

$$\neg X_1 \vee \neg X_2 \vee \neg X_3$$
$$X_1 \vee X_2 \vee \neg X_3$$

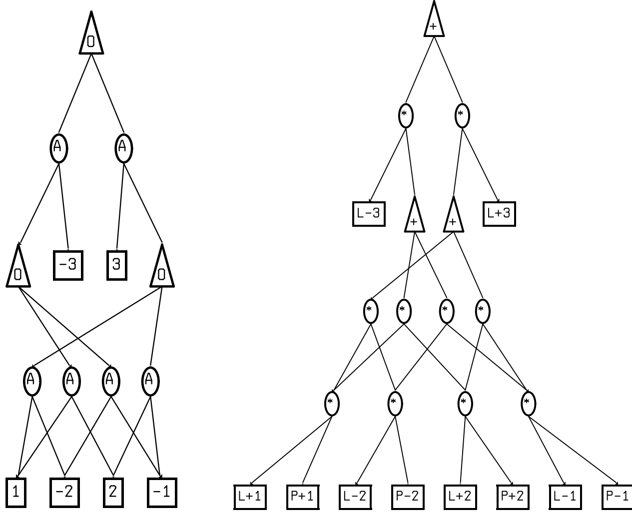


Figure 4: On the left: a smooth d-DNNF which encodes a deterministic belief network $X_1 \rightarrow X_3 \leftarrow X_2$, where X_3 is an exclusive–or of X_1 and X_2. On the right: an arithmetic circuit extracted from the d-DNNF. Here, L+i represents λ_{x_i}, L-i represents $\lambda_{\bar{x_i}}$, P+i represents θ_{x_i}, and P-i represents $\theta_{\bar{x_i}}$.

$$\neg X_1 \vee X_2 \vee X_3$$
$$X_1 \vee \neg X_2 \vee X_3$$

A smooth d-DNNF for this encoding is given in Figure 4(a), and the arithmetic circuit extracted from this d-DNNF is given in Figure 4(b). Let us now see how this circuit can be used to answer queries. Suppose the distribution over input X_1 is $\theta_{x_1} = .4, \theta_{\bar{x_1}} = .6$, and the distribution over input X_2 is $\theta_{x_2} = .7, \theta_{\bar{x_2}} = .3$. Let us now compute the probability $Pr(X_3 = \bar{x_3})$. We can do this by evaluating the circuit under the following inputs $\lambda_{x_1} = \lambda_{\bar{x_1}} = \lambda_{x_2} = \lambda_{\bar{x_2}} = \lambda_{\bar{x_3}} = 1$ and $\lambda_{x_3} = 0$. The circuit evaluates to .46 under these inputs, which is the probability of $X_3 = \bar{x_3}$. One can answer a number of other queries based on the derivatives of the circuit, but we refer the reader to [7, 6] for details.

We now turn to some real–world deterministic belief networks, which correspond to combinational and sequential circuits selected from the ISCAS85 and ISCAS89 suites [1].[7] Table 1 reports on the sizes of d-DNNFs we were able to obtain for the CNF encodings of some members of these suites—more details

[7] Sequential circuits have been converted into combinational circuits in a standard way, by cutting feedback loops into flip-flops, treating a flip-flop's input as a circuit output and its output as a circuit input.

Network	Vars#	d-DNNF nodes#	d-DNNF edge#	Max Clique	d-DNNF Time (s)
c432	196	2899	19779	28	6
c499	243	691803	2919960	23	448
c880	443	3975728	7949684	24	1893
c1355	587	338959	3295293	23	809
c1908	913	6183489	12363322	45	5712
s510	236	967	5755	38	2
s953	440	11542	110266	64	14
s967	439	20645	443233	60	117
s1196	561	12554	261402	51	60
s1238	540	14512	288143	53	58
s1488	667	6338	62175	49	11
s1494	661	6827	64888	51	12
s1512	866	12560	140384	21	27
s3330	1961	358093	8889410	43	5853
s3384	1911	44487	392223	17	45

Table 1: Experimental results on deterministic belief networks.

can be found in [10]. An arithmetic circuit can be extracted from a given d-DNNF by first smoothing the d-DNNF, which increases its size slightly, and then applying the method of Definition 5 in a straightforward manner.[8] Some of the circuits in Table 1 are notorious for their high connectivity [14]. Table 1 lists the circuits we experimented with, in addition to the size of largest clique in the best jointree we could construct for their corresponding belief networks, using both classical methods [20] and more recent ones [11]. As is obvious from the reported clique sizes, these networks are quite hard for structure–based algorithms, such as the jointree algorithm. Moreover, some of them are completely outside the scope of these algorithms, since their complexity is exponential in clique sizes.

The success of our proposed approach on deterministic belief networks appears to be consistent with findings reported in [22], where OBDDs were used to encode a special class of belief networks that arise in troubleshooting applications. The approach in [22] was found to be quite favorable when compared with two implementations of the jointree algorithm: the Shenoy–Shafer [29] and the HUGIN architectures [28]. We note, however, that a number of the CNF encodings corresponding to networks in Table 1 could not be compiled into OBDDs, using the state–of–the–art CUDD package (*http://vlsi.colorado.edu/~fabio/CUDD/*), even though they were compiled successfully into d-DNNFs [10].

6 Conclusion

We have recently proposed a tractable logical form, known as deterministic, decomposable negation nor-

[8] To smooth a d-DNNF, we visit each or–node n and each of its children c. If $Vars(n) \neq Vars(c)$, we replace the child c of n by a new child: $c \wedge \bigwedge_{X \in Vars(n) - Vars(c)} (X \vee \neg X)$. Smoothing is an equivalence–preserving operation.

mal form (d-DNNF). We have shown that d-DNNF supports a number of logical operations in polynomial time, including clausal entailment, model counting, model enumeration, model minimization, and probabilistic equivalence testing. In this paper, we discussed another major application of this logical form: the implementation of multi-linear functions (of exponential size) using arithmetic circuits (that are not necessarily exponential). Specifically, we showed that each multi–linear function can be encoded using a propositional theory, and that each d-DNNF of the theory corresponds to an arithmetic circuit that implements the encoded multi–linear function. We discussed the application of these results to factoring belief networks, which can be viewed as multi–linear functions as has been shown recently. We also discussed the merits of the proposed approach for factoring belief networks, and presented experimental results showing how it can handle efficiently belief networks that are intractable to structure–based methods for probabilistic inference.

Acknowledgement

This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

[1] ISCAS Benchmark Circuits, http://www.cbl.ncsu.edu/www/CBL_Docs.

[2] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ACM/IEEE International Conference on Computer Aided Design*, 1993.

[3] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context–specific independence in bayesian networks. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 115–123, 1996.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[5] B.D. D'Ambrosio. Local Expression Languages for Probabilistic Dependence. *International Journal of Approximate Reasoning*, 13(1):61–81, 1995.

[6] Adnan Darwiche. A differential approach to inference in Bayesian networks. Technical Report D–108, Computer Science Department, UCLA, Los Angeles, Ca 90095, 1999. To appear in the Journal of ACM.

[7] Adnan Darwiche. A differential approach to inference in Bayesian networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 123–132, 2000.

[8] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):1–40, 2001.

[9] Adnan Darwiche. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[10] Adnan Darwiche. A compiler for deterministic decomposable negation normal form. Technical Report D–125, Computer Science Department, UCLA, Los Angeles, Ca 90095, 2002.

[11] Adnan Darwiche and Mark Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*, pages 180–191. Springer-Verlag, 2001.

[12] Adnan Darwiche and Jinbo Huang. Testing equivalence probabilistically. Technical Report D–123, Computer Science Department, UCLA, Los Angeles, Ca 90095, 2002.

[13] Adnan Darwiche and Pierre Marquis. A perspective on knowledge compilation. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 175–182, 2001.

[14] Yousri El Fattah and Rina Dechter. An evaluation of structural paramters for probabilistic reasoning: Results on benchmark circuits. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 244–251, 1996.

[15] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[16] Jesse Hoey, Robert St-Aubin, Alan Hu, and Graig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288, 1999.

[17] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[18] F. V. Jensen, S.L. Lauritzen, and K.G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[19] Frank Jensen and Stig K. Andersen. Approximations in Bayesian belief universes for knowledge based systems. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 162–169, Cambridge, MA, July 1990.

[20] U. Kjaerulff. Triangulation of graphs—algorithms giving small total state space. Technical Report R-90-09, Department of Mathematics and Computer Science, University of Aalborg, Denmark, 1990.

[21] Z. Li and B.D. D'Ambrosio. Efficient Inference in Bayes Networks as a Combinatorial Optimization Problem. *International Journal of Approximate Reasoning*, 11(1):55–81, 1994.

[22] T. Nielsen, P. Wuillemin, F. Jensen, and U. Kjaerulff. Using ROBDDs for inference in Bayesian networks with troubleshooting as an example. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 426–435, 2000.

[23] James Park and Adnan Darwiche. A differential semantics for jointree algorithms. Technical Report D–118, Computer Science Department, UCLA, Los Angeles, Ca 90095, 2001.

[24] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[25] Judea Pearl. *Causality*. Cambridge University Press, 2000.

[26] David Poole. Context-specific approximation in probabilistic inference. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 447–454, 1998.

[27] Graz Rote. Path problems in graphs. *Computing Suppl.*, 7:155–189, 1990.

[28] F. Jensen S. Anderson, K. Olesen and F. Jensen. Hugin – a shell for building beliefs universes for expert systems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 1989.

[29] Prakash P. Shenoy and Glenn Shafer. Propagating belief functions with local computations. *IEEE Expert*, 1(3):43–52, 1986.

[30] J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comp. Sci.*, 3:317–347, 1988.

[31] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.

A Proofs

Proof of Theorem 1

We first define the function *Decode1* for a propositional theory Δ over variables Σ:

$$\mathit{Decode1}(\Delta, \Sigma) \stackrel{def}{=} \sum_{\sigma \models \Delta} \prod_{V \in \Sigma} \begin{cases} 1, & \text{if } \sigma(V) = \mathit{false}; \\ V, & \text{if } \sigma(V) = \mathit{true}. \end{cases}$$

Clearly, $\mathit{Decode1}(\Delta, \Sigma)$ returns the multi–linear function encoded by Δ according to Definition 1.

We now have the following properties of *Decode1*.

Determinism and Smoothness. When $\alpha \wedge \beta$ is inconsistent and $\mathit{Vars}(\alpha) = \mathit{Vars}(\beta) = \Sigma$:

$$\mathit{Decode1}(\alpha \vee \beta, \Sigma) = \mathit{Decode1}(\alpha, \Sigma) + \mathit{Decode1}(\beta, \Sigma).$$

This follows immediately from the definition of *Decode1*.

Decomposability. When $\mathit{Vars}(\alpha) \cap \mathit{Vars}(\beta) = \emptyset$ and $\mathit{Vars}(\alpha) \cup \mathit{Vars}(\beta) = \Sigma$:

$$\begin{aligned} &\mathit{Decode1}(\alpha \wedge \beta, \Sigma) \\ &\quad = \quad \mathit{Decode1}(\alpha, \mathit{Vars}(\alpha)) \star \mathit{Decode1}(\beta, \mathit{Vars}(\beta)). \end{aligned}$$

If f_α is the multi–linear function encoded by α, and if f_β is the multi-linear function encoded by β, then $f_\alpha \star f_\beta$ is the multi–linear function encoded by $\alpha \wedge \beta$ in this case.

Negative literal. $\mathit{Decode1}(\neg V, \{V\}) = 1$. This follows because $\neg V$ has only one model which sets its only variable V to *false*. Hence, $\neg V$ encodes a multi–linear function f that contains one term that has no variables; $f = 1$.

Positive literal. $\mathit{Decode1}(V, \{V\}) = V$. This follows because V has only one model which sets its only variable V to *true*. Hence, V encodes a multi–linear function that contains one term V; $f = V$.

The arithmetic circuit given in Definition 2 is then a trace of the application of function *Decode1* to $\Delta(n)$, where n is the root of smooth d-DNNF Δ. □

Proof of Theorem 2

We prove that the described CNF Δ does indeed encode the multi–linear function f of the given belief

network by showing a one–to–one correspondence between the models of Δ and the terms of f.

Suppose that t is a term in the multi–linear function f. Let σ be the model that encodes t as given by Definition 1. We want to show that $\sigma \models \Delta$. The term t contains exactly one indicator λ_x for each variable X, where x is a value of X. Moreover, it contains exactly one parameter $\theta_{x|u_1,\ldots,u_m}$ for variable X, where the indicators $\lambda_x, \lambda_{u_1}, \ldots, \lambda_{u_m}$ appear in term t. The model σ will then set the indicator λ_x to *true* and all other indicators of X to *false*. It will also set the parameter $\theta_{x|u_1,\ldots,u_m}$ to *true* and all other parameters of X to *false*. The model σ will then clearly satisfy clauses 1–5 and, hence, $\sigma \models \Delta$.

Suppose now that we have a model $\sigma \models \Sigma$, and let t be the term encoded by σ as given by Definition 1. We want to show that t is a term in the multi–linear function f. Since the model σ satisfies clauses 1–2, it must then set exactly one indicator λ_x to true for every variable X. The term t will then include exactly one indicator for each variable X. Moreover, since the model σ satisfies clauses 3–5, it must set to true exactly one parameter for X: $\theta_{x|u_1,\ldots,u_m}$, where $\lambda_x, \lambda_{u_1}, \ldots, \lambda_{u_m}$ are the indicators of $X, U_1, \ldots, U_m$ that are set to true by σ. Hence, $\theta_{x|u_1,\ldots,u_m}$ will be the only parameter of X which is included in the term t. Hence, t is a term in the multi–linear function f. □

Proof of Theorem 3

First, note that the multi–linear function of given belief network has 2^n terms corresponding to the instantiations of network variables. A number of these terms, however, are multiplied by zero parameters and can be excluded. Each term which is not multiplied by a zero conditional parameter, has the form: $\lambda_{x_1}\theta_{x_1} \ldots \lambda_{x_m}\theta_{x_m}\lambda_{x_{m+1}} \ldots \lambda_{x_n}$, where $X_1, \ldots, X_m$ are the root variables in the belief network. This follows since all conditional parameters that are consistent with such a term must be equal to 1.

Let us now define the function *Decode2* for a propositional theory Δ over variables Σ and variables $\Gamma \subseteq \Sigma$:

$$\begin{aligned} &\mathit{Decode2}(\Delta, \Sigma, \Gamma) \\ &\stackrel{def}{=} \sum_{\sigma \models \Delta} \prod_{X \in \Sigma - \Gamma} \begin{cases} \lambda_x, & \text{if } \sigma(X) = \mathit{true} \\ \lambda_{\bar{x}}, & \text{if } \sigma(X) = \mathit{false} \end{cases} \\ &\qquad \prod_{X \in \Gamma} \begin{cases} \lambda_x\theta_x, & \text{if } \sigma(X) = \mathit{true} \\ \lambda_{\bar{x}}\theta_{\bar{x}}, & \text{if } \sigma(X) = \mathit{false}. \end{cases} \end{aligned}$$

If Δ is the CNF encoding a deterministic belief network N, and if N has variables Σ and root variables Γ, then *Decode2*(Δ, Σ, Γ) is clearly the multi–linear function of network N. This follows because the models of Δ are in one–to–one correspondence with the non–vanishing terms of the multi–linear function of N.

We now have the following properties of *Decode2*.

Determinism and Smoothness. When $\alpha \wedge \beta$ is inconsistent and $\mathit{Vars}(\alpha) = \mathit{Vars}(\beta) = \Sigma$:

$$\begin{aligned} &\mathit{Decode2}(\alpha \vee \beta, \Sigma, \Gamma) \\ &\quad = \mathit{Decode2}(\alpha, \Sigma, \Gamma) + \mathit{Decode2}(\beta, \Sigma, \Gamma). \end{aligned}$$

Decomposability. When $\mathit{Vars}(\alpha) \cap \mathit{Vars}(\beta) = \emptyset$ and $\mathit{Vars}(\alpha) \cup \mathit{Vars}(\beta) = \Sigma$:

$$\begin{aligned} &\mathit{Decode2}(\alpha \wedge \beta, \Sigma, \Gamma) \\ &\quad = \mathit{Decode2}(\alpha, \mathit{Vars}(\alpha), \Gamma \cap \mathit{Vars}(\alpha)) \star \\ &\qquad \mathit{Decode2}(\beta, \mathit{Vars}(\beta), \Gamma \cap \mathit{Vars}(\beta)). \end{aligned}$$

Negative literal.

$$\mathit{Decode2}(\neg V, \{V\}, \Gamma) = \begin{cases} \lambda_{\bar{x}}\theta_{\bar{x}}, & \text{if } X \in \Gamma; \\ \lambda_{\bar{x}}, & \text{otherwise.} \end{cases}.$$

Positive literal.

$$\mathit{Decode2}(V, \{V\}, \Gamma) = \begin{cases} \lambda_x\theta_x, & \text{if } X \in \Gamma; \\ \lambda_x, & \text{otherwise.} \end{cases}.$$

The arithmetic circuit given in Definition 5 is then a trace of the application of function *Decode2* to $\Delta(n)$, where n is the root of smooth d-DNNF Δ. □