Compiling Probabilistic Graphical Models using Sentential Decision Diagrams

Arthur Choi, Doga Kisa, and Adnan Darwiche

University of California, Los Angeles, California 90095, USA
{aychoi,doga,darwiche}@cs.ucla.edu

Abstract. Knowledge compilation is a powerful approach to exact inference in probabilistic graphical models, which is able to effectively exploit determinism and context-specific independence, allowing it to scale to highly connected models that are otherwise infeasible using more traditional methods (based on treewidth alone). Previous approaches were based on performing two steps: encode a model into CNF, then compile the CNF into an equivalent but more tractable representation (d-DNNF), where exact inference reduces to weighted model counting. In this paper, we investigate a bottom-up approach, that is enabled by a recently proposed representation, the Sentential Decision Diagram (SDD). We describe a novel and efficient way to encode the factors of a given model directly to SDDs, bypassing the CNF representation. To compile a given model, it now suffices to conjoin the SDD representations of its factors, using an `apply` operator, which d-DNNFs lack. Empirically, we find that our simpler approach to knowledge compilation is as effective as those based on d-DNNFs, and at times, orders-of-magnitude faster.

1 Introduction

There are a variety of algorithms for performing exact inference in probabilistic graphical models; see, e.g., [5, 12]. They typically have time complexities that are exponential in the treewidth of a given model, which make them unsuitable for models with high treewidths. Another approach to exact probabilistic inference, known as *knowledge compilation,* is capable of exploiting local structure in probabilistic graphical models, such as determinism and context-specific independence, allowing one to conduct exact inference efficiently, even in models with high treewidth. The basic idea is to encode then compile a given model into a target representation, where local structure can be exploited in more natural ways. Exact inference then reduces to weighted model counting, where the complexity of inference is now just linear in the size of the representation found [4, 2]. The challenge is then to find effective encodings of a probabilistic graphical model that can be efficiently compiled to representations of manageable size.

Previous approaches based on knowledge compilation can be summarized as performing two steps [4, 2]. First, a given model is encoded as a CNF, where exact inference corresponds to weighted model counting in the CNF. Second, this CNF is compiled into a more tractable representation called deterministic,

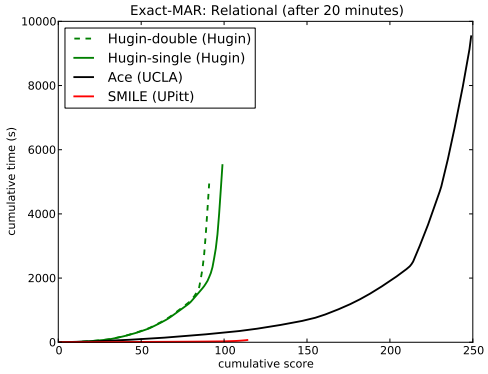


Fig. 1. The performance of ACE at the UAI'08 evaluation of probabilistic reasoning systems, the only system to exactly solve all 250 benchmarks in the `relational` suite.

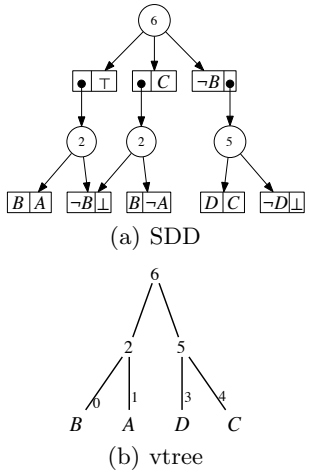


Fig. 2. An SDD and vtree for $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

decomposable negation normal form (d-DNNF) [8]. The effectiveness of this approach depends critically on (1) how we encode a model as a CNF, and (2) on how we compile the CNF into d-DNNF. The ACE system implements this approach, using the C2D system to compile a CNF into a d-DNNF.[1] Figure 1 illustrates the performance benefits of ACE, at the UAI'08 evaluation [7] on an example suite of high treewidth networks that are synthesized from relational models. The average cluster size for this suite is greater than 50, making them infeasible to solve without exploiting local structure.

We propose here a simpler bottom-up approach for compiling probabilistic graphical models, that is enabled by a recently proposed representation, the Sentential Decision Diagram (SDD) [6]. Unlike d-DNNFs, an efficient `apply` operation is available for SDDs, which allows one to conjoin and disjoin two SDDs efficiently. This allows us to bypass intermediate representations in CNF, and encode the factors of a model directly to SDDs. Compilation then reduces to conjoining factors together, as SDDs, also using the `apply` operator. Encoding and compilation are now expressed in common terms, enabling novel, more efficient ways to exploit local structure. Empirically, this leads to a more efficient compilation algorithm, sometimes by orders-of-magnitude. In the process, we propose further a new cardinality minimization algorithm for SDDs.

2 Probabilistic Inference as Weighted Model Counting

We first review how to reduce inference in probabilistic graphical models to weighted model counting, as in [2]. As a running example, we use a simple

[1] ACE is available at `http://reasoning.cs.ucla.edu/ace/`, and C2D is available at `http://reasoning.cs.ucla.edu/c2d/`.

Bayesian network $A \rightarrow B$, where variable A has 2 states a and $\bar{a}$, and variable B has 2 states b and $\bar{b}$. This network has two CPTs, a CPT Θ_A with 2 parameters θ_a and $\theta_{\bar{a}}$, and a CPT $\Theta_{B|A}$ with 4 parameters $\theta_{b|a}, \theta_{\bar{b}|a}, \theta_{b|\bar{a}}$ and $\theta_{\bar{b}|\bar{a}}$.

We can encode a Bayesian network as a propositional knowledge base Δ represented in CNF, whose weighted model count will correspond to the probability of evidence in a Bayesian network: $Pr(\mathbf{e}) = \sum_{\mathbf{x} \sim \mathbf{e}} \prod_X \theta_{x|\mathbf{u}}$, where $\mathbf{x}$ is a complete network instantiation, $\mathbf{e}$ is an evidence instantiation, and relation $\sim$ denotes compatibility between two instantiations (they agree on the values of common variables). For probabilistic graphical models in general, weighted model counts correspond to partition functions.

We first define the propositional variables of the CNF. First, for each BN variable X we define *indicator variables* I_x of the CNF, one variable I_x for each value x of BN variable X. Second, for each CPT $\Theta_{X|\mathbf{U}}$ of our BN, we define *parameter variables* $P_{x|\mathbf{u}}$, one variable $P_{x|\mathbf{u}}$ for each CPT parameter $\theta_{x|\mathbf{u}}$. In our running example, we have the CNF variables:

BN variables	CNF variables
A	$I_a, I_{\bar{a}}$
B	$I_b, I_{\bar{b}}$

BN CPTs	CNF variables
Θ_A	$P_a, P_{\bar{a}}$
$\Theta_{B|A}$	$P_{b|a}, P_{\bar{b}|a}, P_{b|\bar{a}}, P_{\bar{b}|\bar{a}}$

We have two types of clauses in our CNF. First, for each BN variable, we have *indicator clauses*, which enforce a constraint that exactly one of the corresponding indicator variables is true. For each CPT, we have *parameter clauses*, which, given the indicator clauses, enforce a constraint that exactly one of the corresponding parameter variables is true (the one consistent with the indicator variables). In our example, we thus have the clauses:[2]

BN variables	CNF clauses	
A	$I_a \vee I_{\bar{a}}$	$\neg I_a \vee \neg I_{\bar{a}}$
B	$I_b \vee I_{\bar{b}}$	$\neg I_b \vee \neg I_{\bar{b}}$

BN CPTs	CNF clauses	
Θ_A	$I_a \Leftrightarrow P_a$	$I_{\bar{a}} \Leftrightarrow P_{\bar{a}}$
$\Theta_{B|A}$	$I_a \wedge I_b \Leftrightarrow P_{b|a}$	$I_a \wedge I_{\bar{b}} \Leftrightarrow P_{\bar{b}|a}$
	$I_{\bar{a}} \wedge I_b \Leftrightarrow P_{b|\bar{a}}$	$I_{\bar{a}} \wedge I_{\bar{b}} \Leftrightarrow P_{\bar{b}|\bar{a}}$

To do weighted model counting, we need to specify weights on each CNF literal. For each indicator variable, we set both literal weights $W(I_x)$ and $W(\neg I_x)$ to one. For each parameter variable, we set the positive literal weight $W(P_{x|\mathbf{u}})$ to the value of the corresponding BN parameter $\theta_{x|\mathbf{u}}$, and the negative literal weight $W(\neg P_{x|\mathbf{u}})$ to one. The models w of the resulting knowledge base Δ are now in one-to-one correspondence with rows of the joint distribution table induced by our BN. The weight of a model w is $W(w) = \prod_{w \models \ell} W(\ell)$, and the weighted model count of Δ is $\mathsf{wmc}(\Delta) = \sum_{w \models \Delta} W(w)$. For example, we have the following model w and model weight $W(w)$:

$$w = (I_a, \neg I_{\bar{a}}, \neg I_b, I_{\bar{b}}, P_a, \neg P_{\bar{a}}, \neg P_{b|a}, P_{\bar{b}|a}, \neg P_{b|\bar{a}}, \neg P_{\bar{b}|\bar{a}})$$
$$W(w) = W(P_a) \cdot W(P_{\bar{b}|a}) = \theta_a \cdot \theta_{\bar{b}|a} = Pr(a, \bar{b}).$$

[2] $I_a \wedge I_b \Leftrightarrow P_{b|a}$ is shorthand for the clauses $(\neg I_a \vee \neg I_b \vee P_{b|a}), (I_a \vee \neg P_{b|a})$ $(I_b \vee \neg P_{b|a})$.

Further, the weighted model count is one, just as a BN's joint probability table sums to one. We incorporate evidence by setting to zero the weights of any indicator variable I_x that is not compatible with the evidence. The weighted model count then corresponds to the probability of evidence in a BN.

2.1 Exploiting Local Structure

Zero Parameters It is straightforward to encode determinism using CNFs. Say that the parameter $\theta_{b|a}$ is zero. Any model w where parameter variable $P_{b|a}$ appears positively has a model weight that is zero, since $W(P_{b|a}) = 0$. We can thus replace the parameter clauses for the BN parameter $\theta_{x|\mathbf{u}}$ with the single clause $\neg I_a \vee \neg I_b$, which also forces to zero the weight of a model where parameter variable $P_{b|a}$ appears positively. The parameter variable $P_{b|a}$ is now superfluous, and can be removed from the domain of the knowledge base Δ.

Equal Parameters Efficiently encoding equal parameters can be a more subtle process. Say that two parameters of a CPT, say $\theta_{b|a}$ and $\theta_{\bar{b}|\bar{a}}$, have the same value p. The parameter clauses (given the indicator clauses) guarantee that exactly one parameter variable from each CPT appears positively in any model $w \models \Delta$. This allows us to use a common parameter variable P_p, for equal parameters. If these parameters have clauses $I_a \wedge I_b \Leftrightarrow P_{b|a}$ and $I_{\bar{a}} \wedge I_{\bar{b}} \Leftrightarrow P_{\bar{a}|\bar{b}}$ we first instead assert the clauses $I_a \wedge I_b \Rightarrow P$ and $I_{\bar{a}} \wedge I_{\bar{b}} \Rightarrow P$. This by itself is not sufficient, as the resulting knowledge base Δ admits too many models (the above clauses, by themselves, do not prevent parameter variable P from being set to true when neither $I_a \wedge I_b$ nor $I_{\bar{a}} \wedge I_{\bar{b}}$ are true). We can filter out such models once we compile the resulting CNF into d-DNNF, by performing cardinality minimization [2].

3 Sentential Decision Diagrams

The Sentential Decision Diagram (SDD) is a newly introduced target representation for propositional knowledge bases [6]. It is a strict subset of deterministic, decomposable negation normal form (d-DNNF), used by the ACE system. Figure 2(a) depicts an SDD: paired-boxes $\boxed{p\,|\,s}$ are called *elements* and represent conjunctions $(p \wedge s)$, where p is called a *prime* and s is called a *sub*. Circles are called *decision nodes* and represent disjunctions of the corresponding elements. SDDs satisfy stronger properties than d-DNNFs, allowing one, for example, to conjoin two SDDs in polytime. In contrast, this is not possible in general with d-DNNFs [8]. As we shall show, the ability to conjoin SDDs efficiently is critical for incremental, bottom-up compilation of probabilistic graphical models.

An SDD is constructed for a given *vtree,* which is a full binary tree whose leaves are in one-to-one correspondence with the given variables; see Figure 2(b). The SDD is canonical for a given vtree (under some conditions) and its size depends critically on the vtree used. Ordered Binary Decision Diagrams (OBDDs) [1] are a strict subset of SDDs: OBDDs correspond precisely to SDDs that are constructed using a special type of vtree, called a right-linear vtree [6].

Theoretically, SDDs come with size upper bounds (based on treewidth) [6] that are tighter than the size upper bounds that OBDDs come with (based on pathwidth) [13, 11, 9]. In practice, dynamic compilation algorithms can find SDDs that are orders-of-magnitude more succinct than those found using OBDDs [3]. Compilation to d-DNNF has also compared favorably against bottom-up compilation using OBDDs in other probabilistic representations [10].

Every decision node in an SDD is *normalized* for some vtree node. In Figure 2(a), each decision node is labeled with the vtree node it is normalized for. Consider a decision node with elements $\boxed{p_1 \mid s_1}, \ldots, \boxed{p_n \mid s_n}$, and suppose that it is normalized for a vtree node v which has variables $\mathbf{X}$ in its left subtree and variables $\mathbf{Y}$ in its right subtree. We are then guaranteed that each prime p_i will only mention variables in $\mathbf{X}$ and that each sub s_i will only mention variables in $\mathbf{Y}$ (this ensures decomposability). Moreover, the primes are guaranteed to represent propositional sentences that are consistent, mutually exclusive, and exhaustive (this ensures determinism). For example, the top decision node in Figure 2(a) has elements that represent the following sentences:

$$\{(\underbrace{A \wedge B}_{\text{prime}}, \underbrace{\mathsf{true}}_{\text{sub}}), (\underbrace{\neg A \wedge B}_{\text{prime}}, \underbrace{C}_{\text{sub}}), (\underbrace{\neg B}_{\text{prime}}, \underbrace{D \wedge C}_{\text{sub}})\}$$

One can verify that these primes and subs satisfy the properties above.

In our experiments, we use the SDD package developed by the Automated Reasoning Group at UCLA.[3] This package allows one to efficiently conjoin, disjoin and negate SDDs, in addition to computing weighted model counts in time that is linear in the size of the corresponding SDD.

4 Bottom-Up Compilation Into SDDs

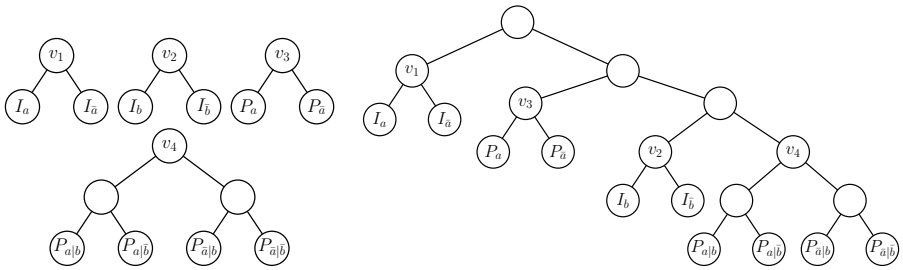
There are a number of steps we need to take in order to compile a given probabilistic graphical model into an equivalent representation as an SDD. At each step, we make certain decisions that can have a significant impact on the size of the resulting SDD, as well as on the efficiency of constructing it.

4.1 Choosing an Initial Vtree

Vtrees uniquely define SDDs (under some conditions), so the choice of an initial vtree is critical to obtaining a compact SDD. Here, we consider one approach, which we describe below, that was effective in our experiments.

We propose to obtain an initial vtree by inducing one from a variable ordering. We run the min-fill algorithm on a given model, and use the resulting variable ordering to induce a vtree, as follows. First, we construct for each model variable, a balanced vtree over indicator variables, and for each factor, a balanced vtree over parameter variables. We then simulate variable elimination: (1) when we multiply two factors, we compose their vtrees, and (2) when we forget

[3] Publicly available at `http://reasoning.cs.ucla.edu/sdd`.



(a) Initial Balanced Vtrees (b) Vtree for Network $A \rightarrow B$

Fig. 3. Choosing an initial vtree for network $A \rightarrow B$

a variable from a factor, we compose the vtrees of the variable and the factor. Here, composing two vtrees v_i and v_j means that we create a new vtree v with children v_i and v_j. In our experiments, we let the left subtree be the one with fewer variables. Figure 3(a) shows the initial vtrees over indicator and parameter variables for a Bayesian network $A \rightarrow B$. Figure 3(b) shows a vtree constructed using variable ordering $\langle B, A \rangle$. First, forget variable B from CPT $\Theta_{B|A}$ (compose vtrees v_2 and v_4), then multiply with CPT Θ_A (compose with vtree v_3), and finally forget variable A (compose with vtree v_1).

4.2 Compiling Factors Into SDDs

Here, we consider how to efficiently encode the factors of a given model as an SDD. This encoding is possible as SDDs support an efficient `apply` operator, which given two SDDs α and β and a boolean operator $\circ$, will return a new SDD for $\alpha \circ \beta$, in polytime. When we encode a factor, we encode the indicator clauses for each factor variable, and the parameter clauses for each factor parameter, as in Section 2. This factor CNF can be compiled using `apply`, where we disjoin the corresponding literals of each clause, and then conjoin the resulting clauses. However, we could seek a more direct and efficient approach by relaxing our use of the CNF representation. Consider the factor $\Theta_{B|A}$ of network $A \rightarrow B$, with parameters $\theta_{b|a}, \theta_{\bar{b}|a}, \theta_{b|\bar{a}}$ and $\theta_{\bar{b}|\bar{a}}$. Our factor CNF is equivalent to the following DNF, over indicator variables I_a and I_b and parameter variables $P_{b|a}$:

$$\begin{aligned}
&(I_a \wedge \neg I_{\bar{a}} \wedge I_b \wedge \neg I_{\bar{b}} \wedge P_{b|a} \wedge \neg P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}} \wedge \neg P_{\bar{b}|\bar{a}}) \\
\vee\ &(I_a \wedge \neg I_{\bar{a}} \wedge \neg I_b \wedge I_{\bar{b}} \wedge \neg P_{b|a} \wedge P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}} \wedge \neg P_{\bar{b}|\bar{a}}) \\
\vee\ &(\neg I_a \wedge I_{\bar{a}} \wedge I_b \wedge \neg I_{\bar{b}} \wedge \neg P_{b|a} \wedge \neg P_{\bar{b}|a} \wedge P_{b|\bar{a}} \wedge \neg P_{\bar{b}|\bar{a}}) \\
\vee\ &(\neg I_a \wedge I_{\bar{a}} \wedge \neg I_b \wedge I_{\bar{b}} \wedge \neg P_{b|a} \wedge \neg P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}} \wedge P_{\bar{b}|\bar{a}}).
\end{aligned}$$

The constraints implied by indicator and parameter clauses result in a DNF where each term represents a setting of indicator and parameter variables for each factor parameter $\theta_{x|\mathbf{u}}$. In particular, the term for parameter $\theta_{x|\mathbf{u}}$ has positive literals for parameter variable $P_{x|\mathbf{u}}$ and for the indicator variables consistent with instantiation $x\mathbf{u}$; all other literals are negative.

Using `apply`, it is also easy to compile a DNF into an SDD. However, there are as many terms in the DNF as there are parameters in a factor, and each term has a sub-term with the same number of literals. Naively, this entails a quadratic number of `apply` operations, just to construct the sub-terms over parameter variables, which is undesirable when we have large factors.

Instead, we observe that each sub-term over parameter variables, has all but one variable appearing negatively. We thus construct an SDD α for the sub-term composed of all negative literals, using `apply`. We can use, and re-use, this SDD to construct all parameter sub-terms, using two additional operations each. In particular, for each parameter $\theta_{x|\mathbf{u}}$ we compute $(\alpha \mid \neg P_{x|\mathbf{u}}) \wedge P_{x|\mathbf{u}}$, where $(\alpha \mid \ell)$ denotes conditioning α on a literal ℓ (which is another operation supported by SDDs). This conditioning is equivalent to replacing $\neg P_{x|\mathbf{u}}$ with the constant true, which drops the literal from the term α. The conjoin then replaces the literal with the positive one. To construct all sub-terms over parameter variables, we just need in total a linear number of `apply` operations and a linear number of conditioning operations, which is much more efficient than a quadratic number of `apply`'s. The same technique can be used to construct terms over indicator variables, which is similarly effective when a factor contains variables with many states. We can then conjoin the indicator sub-term with the parameter sub-term.

Encoding Determinism If a factor contains a zero parameter $\theta_{x|\mathbf{u}}$, then any model w satisfying that term, in the factor DNF, will evaluate to zero, since $W(P_{x|\mathbf{u}}) = 0$. Setting variable $P_{x|\mathbf{u}}$ to false does the same, which effectively removes the term and variable from the DNF. The variable $P_{x|\mathbf{u}}$ is now vacuous, so we remove it from the domain of knowledge base Δ.

Encoding Equal Parameters If a factor contains parameters $\theta_{x|\mathbf{u}}$ that have the same value p, then it suffices to have a single parameter variable P_p for those parameters. For example, say parameter $\theta_{b|a}$ and $\theta_{\bar{b}|\bar{a}}$ have the same value p in CPT $\Theta_{B|A}$. The corresponding DNF is:

$$\begin{aligned}
&(I_a \wedge \neg I_{\bar{a}} \wedge I_b \wedge \neg I_{\bar{b}} \wedge P_p \wedge \neg P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}}) \\
\vee\ &(I_a \wedge \neg I_{\bar{a}} \wedge \neg I_b \wedge I_{\bar{b}} \wedge \neg P_p \wedge P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}}) \\
\vee\ &(\neg I_a \wedge I_{\bar{a}} \wedge I_b \wedge \neg I_{\bar{b}} \wedge \neg P_p \wedge \neg P_{\bar{b}|a} \wedge P_{b|\bar{a}}) \\
\vee\ &(\neg I_a \wedge I_{\bar{a}} \wedge \neg I_b \wedge I_{\bar{b}} \wedge P_p \wedge \neg P_{\bar{b}|a} \wedge \neg P_{b|\bar{a}}).
\end{aligned}$$

Note that the weight of each term is unchanged. To compile this function using the `apply` operator, it suffices to construct the parameter sub-term for equal parameters once, and just disjoin the corresponding indicator sub-terms. This is more efficient (fewer `apply` operations) than explicitly compiling the DNF.

Note that in the CNF encoding of Section 2, we resorted to encoding a representation that contained too many models, and then filtered them by performing cardinality minimization after compiling to d-DNNF. This is more efficient than encoding equal parameters directly as a CNF, as a straightforward conversion leads to a CNF with many clauses. However, such techniques are not needed using an SDD representation, as we are not constrained to using CNFs/DNFs.

4.3 Bottom-Up Compilation

Once we have obtained SDD representation of our model's factors, we just need to conjoin these SDDs to obtain an SDD representation of our model. However, what order do we use to conjoin factor SDDs together? This decision impacts the sizes of the intermediate representations that we encounter during compilation. In the implementation we evaluate empirically, we mirror the process we used to construct our vtree, using the same min-fill variable ordering. We start with SDD representations of each factor, and simulate variable elimination: (1) when we multiply two factors, we conjoin the corresponding SDDs, and (2) when we forget a variable from a factor, we conjoin the variable's indicator clauses.[4]

4.4 CNF Encodings: Revisited

Using SDDs, it is also possible to perform bottom-up compilation using the CNF encoding [3]. Suppose we are given a probabilistic graphical model as a set of indicator and parameter clauses, as in Section 2, and a vtree over its indicator and parameter variables, as in Section 4.1. We can assign each clause c to the lowest (and unique) vtree node v which contains its variables. This labeled vtree provides a recursive partitioning of the CNF clauses, with each node v in the vtree hosting a set of clauses Δ_v. To compile a CNF, we recursively compile the clauses placed in the sub-vtrees rooted at the children of v, each child returning with its corresponding SDD. We conjoin these two SDDs using `apply`, and then iterate over the clauses at node v, compiling each into an SDD, and conjoining the result with the existing SDD, all also using `apply`. We also visit the clauses hosted by node v according to their size, visiting shorter clauses first.

4.5 Minimizing Cardinality

When exploiting local structure with a CNF as in Section 2, we appealed to cardinality minimization in a compiled d-DNNF. We need to be able to do the same when compiling CNFs to SDDs, which is not as straightforward.

Formally, the minimum cardinality of a given SDD α is defined as:[5]

$$\texttt{mcard}(\alpha) = \begin{cases} 0 & \text{if } \alpha \text{ is a negative literal or true;} \\ 1 & \text{if } \alpha \text{ is a positive literal;} \\ \infty & \text{if } \alpha \text{ is false.} \\ \min_i\{\texttt{mcard}(p_i) + \texttt{mcard}(s_i)\} & \text{if } \alpha = \{(p_1, s_1), ..., (p_n, s_n)\} \end{cases}$$

Algorithm 1 describes how to recursively obtain an SDD $\alpha_{\min}$ representing the minimum cardinality models of a given SDD α, which we call a minimized SDD.

[4] Conjoining indicator clauses is typically redundant, since we can normally assume they are encoded in the SDD of each factor mentioning that variable.

[5] More intuitively, the cardinality of a model w is the number of positive literals that appear in that model. The minimum cardinality of a knowledge base Δ is the minimum cardinality of all its models. Minimizing a knowledge base Δ produces another knowledge base representing the minimum cardinality models of Δ.

Algorithm 1: `Minimize-SDD`

Input: An SDD α, a vtree v for which α is normalized
Output: A minimized SDD $\alpha_{\min}$, normalized for v, for SDD α

```
1  if α ∈ {⊥, X, ¬X} and v is leaf with variable X then return α;
2  else if α = ⊤ and v is leaf with variable X then return ¬X;
3  else
4      if cache(α) ≠ nil then return cache(α);
5      α_min ← empty decision node;
6      foreach element (p, s) in α do
7          if mcard(p) + mcard(s) > mcard(α) then add (p, ⊥) to α_min;
8          else
9              p_min ← Minimize-SDD(p, v^l), s_min ← Minimize-SDD(s, v^r);
10             add (p_min, s_min) to α_min;
11             p_carry ← apply(p, ¬p_min, ∧);
12             if p_carry ≠ ⊥ then add (p_carry, ⊥) to α_min;
13     add α_min to cache;
14     return α_min;
```

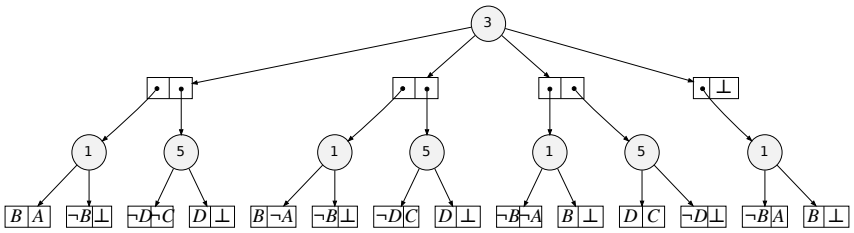


Fig. 4. Minimized SDD for $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$

For each element $(p, s) \in \alpha$, if $\texttt{mcard}(p) + \texttt{mcard}(s) > \texttt{mcard}(\alpha)$, then $(p, \bot)$ is an element of $\alpha_{\min}$. If $\texttt{mcard}(p) + \texttt{mcard}(s) = \texttt{mcard}(\alpha)$, then $(p_{\min}, s_{\min})$ is an element of $\alpha_{\min}$, where $p_{\min}$ and $s_{\min}$ are the minimized SDD's for p and s respectively. If a prime is minimized, then to ensure the exhaustiveness of the primes, we need to add a new element to the minimized SDD (Line 12).

Figure 4 shows the minimized SDD $\alpha_{\min}$ for SDD α in Figure 2. Each element of α has a minimum cardinality of 2, so the minimum cardinality of α is 2. For each element (p, s) of α, the minimization $\alpha_{\min}$ has a minimized element $(p_{\min}, s_{\min})$. The minimization $\{(\neg B,\neg A),(B,\bot)\}$ of prime $\neg B$ is not equal to itself, so $\alpha_{\min}$ has an element with prime $\{(\neg B,A),(B,\bot)\}$ and sub $\bot$.

5 Experiments

We evaluate our approach to compiling probabilistic graphical models (PGMs) into SDDs, where we consider the impact that different encodings can have on

the succinctness of the resulting compilation, and also on the time that it takes to compile it. We compare 4 methods to compile a probabilistic graphical model:[6]

- compiling to SDD without exploiting local structure (denoted by `none`);
- compiling to SDD, exploiting local structure as in Section 4.2 (`sdd`);
- encoding the PGM as a CNF, and compiling to SDD (`cnf`);
- encoding the PGM as a CNF, and compiling to d-DNNF with C2D (`c2d`).

Note that method `c2d` is the one that underlies the ACE system. All methods here are driven by initial structures (vtrees for SDDs and dtrees for d-DNNFs), based on min-fill variable orderings. We restrict ourselves to static initial structures, although SDDs support the ability to dynamically optimize the size of an SDD [3]; top-down approaches to compilation, like method `c2d`, typically do not.

Table 1 highlights statistics for a selection of benchmarks and their SDD and d-DNNF compilations. Here, the size of an SDD is the aggregate size of an SDD's decompositions, and the number of nodes is the number of decision nodes. For methods that compile to SDDs, we observe that encoding local structure (`sdd`, `cnf`) can obtain much more compact SDDs than without (`none`). For example, in network `water`, method `sdd` produced an SDD that was 73× more compact than `none`. Such improvements are typical for knowledge compilation approaches to exact inference, when there is sufficient local structure [2]. Next, methods `cnf` and `sdd` encode the same local structure, so both approaches yield the same compiled SDDs. However, by not constraining ourselves to CNFs, as method `cnf` does, we can obtain these SDDs in much less time, often by orders-of-magnitude.

As for d-DNNFs compiled by C2D, we report the number of NNF edges as the compilation size, and the number of AND-nodes and OR-nodes in an NNF as the number of nodes. While the sizes for SDDs and d-DNNFs are not directly comparable, we note that for instances where we obtained both an SDD and a d-DNNF, the reported sizes are within an order-of-magnitude of each other (or better). This suggests that methods `sdd` and `c2d` are performing comparably, relatively speaking, across these benchmarks. In other cases, method `sdd` could compile benchmarks that method `c2d` was unable to in one hour. For example, method `sdd` was able to compile network `diabetes` in under 25 seconds (at least 144× faster), and method `sdd` was the only one to compile network `munin1`.

Finally, we note that d-DNNFs are in general more succinct than SDDs, and for any given SDD there is a corresponding d-DNNF that is at least as succinct. However, SDDs enjoy an efficient `apply` operator, which is critical for certain applications that are out of the scope of d-DNNFs, which does not support an `apply`. Our results here suggest that our simplified approach (method `sdd`) can be orders-of-magnitude more efficient than other alternatives. In some cases, in fact, the ability to encode directly to an SDD alone (`none`) appears to outweigh the ability to exploit local structure using CNFs (given enough memory).

[6] Our experiments were performed on an Intel i7-3770 3.4GHz CPU with 16GB RAM, except for method `none`, which were on an Intel Xeon E5440 2.83GHz CPU with 32GB RAM (SDD size is the relevant comparison here, and less compilation time).

	PGM stats					compilation stats			
benchmark	X	θ	0	p	C	method	size	nodes	time
barley	48	130180	0	36926	23.4	none	231,784,907	96,062,825	227.46
						sdd	49,442,901	17,678,076	32.48
						cnf	—	—	—
						c2d	—	—	—
diabetes	413	461069	352224	17574	17.5	none	78,641,365	32,312,892	308.74
						sdd	21,704,366	7,882,652	24.49
						cnf	—	—	—
						c2d	—	—	—
diagnose-b	329	34704	51	976	18.1	none	15,622,318	7,115,750	67.23
						sdd	227,170	102,856	0.84
						cnf	227,170	102,856	245.12
						c2d	369,426	124,393	77.56
mildew	35	547158	509234	6713	19.6	none	54,726,909	26,136,443	188.51
						sdd	2,981,951	1,156,072	5.55
						cnf	—	—	—
						c2d	167,676,317	3,120,074	1430.90
munin1	189	19466	10910	4246	26.2	none	⋆	⋆	⋆
						sdd	139,855,161	61,376,880	339.34
						cnf	—	—	—
						c2d	—	—	—
munin2	1003	83920	46606	22852	17.4	none	25,068,547	10,453,726	122.08
						sdd	8,007,175	3,430,400	19.12
						cnf	8,007,175	3,430,400	2377.67
						c2d	—	—	—
munin3	1044	85855	47581	24102	17.3	none	43,069,070	19,066,130	158.25
						sdd	9,623,616	4,431,843	21.73
						cnf	—	—	—
						c2d	66,048,268	2,297,199	132.96
water	32	13484	6970	3578	20.8	none	29,881,265	12,566,205	36.17
						sdd	405,538	195,502	3.83
						cnf	405,538	195,502	106.07
						c2d	1,342,307	141,167	3.24
mm-5-8-3	1616	11278	5531	3367	28.0	none	⋆	⋆	⋆
						sdd	870,867	400,872	255.38
						cnf	870,867	400,872	1830.93
						c2d	4,920,481	214,281	8.78
gr-90-26-1	676	5202	2360	1704	40.0	none	⋆	⋆	⋆
						sdd	600,816	288,632	190.01
						cnf	600,816	288,632	62.48
						c2d	216,935	28,241	3.38

Table 1. Under PGM stats, X is # of variables, θ is # of model parameters, 0 is # of zeros, p is # of distinct parameter values, C is $\log_2$ size of largest jointree cluster. We have 4 compilation methods: none is compilation to SDDs without encoding local structure, sdd is to SDDs with encoding local structure, cnf is to SDDs via encoding CNFs with local structure, c2d is to d-DNNFs via the same CNF using the C2D compiler. Time reported in seconds. — is a 1 hour timeout, ⋆ is out-of-memory.

6 Conclusion

In this paper, we outlined a knowledge compilation approach for exact inference in probabilistic graphical models, that is enabled by a recently proposed representation, the Sentential Decision Diagram (SDD). SDDs support an efficient `apply` operation, which was not available in previous approaches based on compilation to d-DNNFs. As we illustrated, an efficient `apply` operation enables a more unified approach to knowledge compilation, that allows us to encode a model, exploit its local structure, and compile it to a more compact representation, in common and simplified terms. Empirically, we found that by bypassing the auxiliary CNF representations that were previously used, we can obtain SDDs that are of comparable succinctness to d-DNNFs found by C2D, but more efficiently, by orders-of-magnitude in some cases. In the process, we further proposed a new algorithm for minimizing cardinality in SDDs.

Acknowledgments. This work has been partially supported by ONR grant #N00014-12-1-0423, NSF grant #IIS-1118122, and NSF grant #IIS-0916161.

References

1. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
2. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artificial Intelligence Journal 172(6–7), 772–799 (2008)
3. Choi, A., Darwiche, A.: Dynamic minimization of sentential decision diagrams. In: Proceedings of the 27th Conference on Artificial Intelligence (AAAI) (2013)
4. Darwiche, A.: A differential approach to inference in Bayesian networks. Journal of the ACM 50(3), 280–305 (2003)
5. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
6. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: IJCAI. pp. 819–826 (2011)
7. Darwiche, A., Dechter, R., Choi, A., Gogate, V., Otten, L.: Results from the probablistic inference evaluation of UAI-08 (2008), `http://graphmod.ics.uci.edu/uai08/Evaluation/Report`
8. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research 17, 229–264 (2002)
9. Ferrara, A., Pan, G., Vardi, M.Y.: Treewidth in verification: Local vs. global. In: LPAR. pp. 489–503 (2005)
10. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., Raedt, L.D.: Inference in probabilistic logic programs using weighted CNF's. In: UAI. pp. 211–220 (2011)
11. Huang, J., Darwiche, A.: Using DPLL for efficient OBDD construction. In: SAT (2004)
12. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)
13. Prasad, M.R., Chong, P., Keutzer, K.: Why is ATPG easy? In: DAC. pp. 22–28 (1999)