Efficient Symbolic Integration for Probabilistic Inference

Efficient Symbolic Integration for Probabilistic Inference

Samuel Kolb[a*], **Martin Mladenov**[b], **Scott Sanner**[c], **Vaishak Belle**[d], **Kristian Kersting**[e]
[a] KU Leuven [b] Google Research [c] University of Toronto
[d] University of Edinburgh & Alan Turing Institute [e] TU Darmstadt
samuel.kolb@kuleuven.be, mmladenov@google.com, ssanner@mie.utoronto.ca,
vaishak@ed.ac.uk, kersting@cs.tu-darmstadt.de

Abstract

Weighted model integration (WMI) extends weighted model counting (WMC) to the integration of functions over mixed discrete-continuous probability spaces. It has shown tremendous promise for solving inference problems in graphical models and probabilistic programs. Yet, state-of-the-art tools for WMI are generally limited either by the range of amenable theories, or in terms of performance. To address both limitations, we propose the use of extended algebraic decision diagrams (XADDs) as a compilation language for WMI. Aside from tackling typical WMI problems, XADDs also enable partial WMI yielding parametrized solutions. To overcome the main roadblock of XADDs – the computational cost of integration – we formulate a novel and powerful exact symbolic dynamic programming (SDP) algorithm that seamlessly handles Boolean, integer-valued and real variables, and is able to effectively cache partial computations, unlike its predecessor. Our empirical results demonstrate that these contributions can lead to significant computational reduction over existing probabilistic inference algorithms.

1 Introduction

Weighted model counting (WMC) is the problem of computing the mass of a function over the set of models of a propositional theory and lies at the heart of probabilistic artificial intelligence, where a core issue is to quantify uncertainty over logically-structured worlds. Many state-of-the-art algorithms dealing with discrete Bayesian networks [Chavira and Darwiche, 2008], factor graphs [Choi *et al.*, 2013], probabilistic programs [Fierens *et al.*, 2013], and probabilistic databases [Suciu *et al.*, 2011] reduce their inference problem to a WMC computation. While a typical WMC inference task is to compute the partition functions and marginals of factored probability distributions, it has also been used as a subroutine for more general tasks such as automated planning [Domshlak and Hoffmann, 2007].

Many of these successes have been powered by the development of 1) efficient model counting strategies and 2) efficient data structures for manipulating Boolean theories that support WMC. Inference in continuous and mixed discrete-continuous theories, however, had neither enjoyed fast algorithms, nor was it known whether there are efficient data structures for manipulation, until recently. Progress on the algorithmic front accelerated with the introduction of weighted model integration (WMI) [Belle *et al.*, 2015] which extends the usual WMC setting by including real-valued variables, linear real-arithmetic (LRA) theories and symbolic weight functions. WMI (or closely related formulations) have recently been applied to a number of non-trivial graphical modeling and probabilistic programming tasks [Chistikov *et al.*, 2015; Albarghouthi *et al.*, 2017; Morettin *et al.*, 2017; Belle, 2017; Braz *et al.*, 2016]. On the data structure front, an extension of algebraic decision diagrams (ADDs) to piecewise polynomial functions over continuous linear and Boolean theories, so-called *extended ADDs* (XADDs), was proposed in Sanner et al. (2011) and when combined with *symbolic dynamic programming* techniques has proven successful for probabilistic planning as well as probabilistic inference [Sanner and Abbasnejad, 2012]. In the software verification community, a subset of XADDs dealing with piecewise-constant functions was independently developed under the name linear decision diagrams (LDDs) [Chaki *et al.*, 2009].

Although impressive, progress in the mixed discrete-continuous domain on both the algorithmic and data structure front is still far from its Boolean counterpart. Most WMI solvers are based on the so-called block-clause strategy, which naively enumerates the models of a LRA theory and is often prohibitive in practice. Finally, most solvers, including improved variants such as the predication abstraction solver introduced by Morettin et al. (2017), do not consider partial WMI and the efficient storage of intermediate results of computation, which can speed up computing repeated conditional queries, occurring in tasks such as parameter estimation from data. Indeed, Belle *et al.* (2016) considered component caching but is restricted to interval formulas. From the data-structure perspective, current XADD methods suffer from an extremely high cost for integration in WMI; this has severely limited their application to problems with only a handful of variables and formulas [Sanner and Abbasnejad, 2012]. Finally, many probabilistic programming problems, see e.g. the argument of Braz *et al.* (2016), involve Boolean, real-valued but also integer-valued variables, however, both existing state-of-the-art WMI solvers and XADDs currently do not support integer-valued variables. An exception is actually the work of Braz et al., which is probably closest to our work in terms of functionality. It supports both integer and real values, but does not repre-

[*]S. Kolb supported by the Research Foundation-Flanders (FWO)

sent and cache intermediate results, and Morettin et al. (2017) showed that their solver achieved better performance.

In this paper, we address both shortcomings of XADDs and provide additional extensions. Our contribution is centered around a novel and powerful *symbolic dynamic programming algorithm* for marginalization and integration. Concretely, we make the following contributions: **1)** we explore the use of XADDs [Sanner *et al.*, 2011] as a *compilation language* for WMI and show how to structure WMI problems as symbolic dynamic programming over XADDs; **2)** we contribute a novel marginalization and integration algorithm for XADDs that efficiently computes partial integrations, improving the state-of-the-art integration algorithms [Sanner and Abbasnejad, 2012] by exploiting shared substructures in the XADD's directed acyclic graph through caching and we additionally show how to handle integer-valued variables; and **3)** we show that the algorithm can be adapted for volume computations and, moreover, that XADDs can be used to compute partial WMI through partial integration, obtaining parametrized solutions that can be used for repeated query computations or, for example, computing the argmax for the remaining variables.

Our empirical results show that our novel WMI solver can lead to an exponential to linear computational reduction over previous state-of-the-art solvers for problem domains containing high levels of mutually exclusive or XOR structure, and that its performance exceeds or matches existing WMI solvers.

We remark that the decision versions of both #SAT and Bayesian inference are #P-complete. Nonetheless, the caching of partial computations and memoization schemes have been shown to achieve strong time-space tradeoffs, and are often very effective in practice [Bacchus *et al.*, 2009].

2 Background

Foundations of Weighted Model Integration: We assume that the reader is familiar with propositional logic and the SAT problem. Satisfiability Modulo Theories (SMT) generalizes SAT to determining the satisfiability of a formula with respect to a decidable background theory. We consider the background theories LRA and LIA, which are first-order logic fragments restricting the interpretation of numbers, inequalities and operators (sum, product) to their semantics in linear real and integer arithmetic, respectively, as in, e.g., $a \wedge (b \vee (u + v \leq 5))$.

Model counting (#SAT) refers to the task of counting the number of models that satisfy a given formula. Weighted model counting (WMC) generalizes this task by summing the weights assigned to models satisfying a formula. Corresponding to the generalization of SAT to hybrid domains, *weighted model integration (WMI)* generalizes WMC to support SMT(LRA) formulas and real variables [Belle *et al.*, 2015]. We, additionally, extend the WMI formulation to also support linear arithmetic over integers (LIA). Formulas are then Boolean combinations of Boolean literals, inequalities over real variables and inequalities over integer variables, which we will refer to as LA formulas.

Given n real variables $\mathbf{x}$, m Boolean variables A, p integer variables I, an LA formula $\theta(\mathbf{x}, A, I)$ over $\mathbf{x}$, A and I and a weight function $w(\mathbf{x}, A, I)$ that maps variable assignments to real weights, the weighted model integral (WMI), following its most recent definition [Morettin *et al.*, 2017], is defined as (using $WMI(\theta, w)$ to abbreviate $WMI(\theta, w|A, \mathbf{x}, Z)$):

$$WMI(\theta, w) = \sum_{\mu^A \in \mathbb{B}^m} \sum_{\mu^Z \in \mathbb{Z}^p} \int_{\theta(\mathbf{x}, \mu^A, \mu^Z)} w(\mathbf{x}, \mu^A, \mu^Z) d\mathbf{x}$$

That is, the WMI is obtained by summing over every pair of total truth assignments μ^A and μ^Z to the Boolean- and integer variables, substituting every Boolean or integer variable in θ and w with its truth value in μ^A or μ^Z, and integrating $w(\mathbf{x}, \mu^A, \mu^Z)$ over the values $\{\mathbf{x}^* | \theta(\mathbf{x}^*, \mu^A, \mu^Z)\}$.

In order to obtain algorithmic results we restrict w to the class of piecewise-polynomial case functions, i.e.,

$$f = \{\theta_1 : f_1, \ldots, \theta_n : f_n\} \,, \tag{1}$$

where the f_i are polynomial functions over real and integer variables and the case conditions θ_i are LA formulas that partition the underlying space. This class strictly generalizes the class of *polynomial under LRA conditions* functions [Morettin *et al.*, 2017]. As input language for defining polynomial case functions, we fix the language of nested arithmetic and ite compositions, i.e., a single case $\phi = \{\theta : f, \neg\theta : 0\}$ is a case function; if ϕ_1, ϕ_2 are case functions, then so are $\phi_1 + \phi_2$, $\phi_1 * \phi_2$, and, $ite(\theta, \phi_1, \phi_2)$, where θ is a sentence from one of B, LRA or LIA, and ite expands to *if-then-else* understood in the usual manner. This language corresponds to the one implemented in the well-known ***PySMT*** package [Gario and Micheli, 2015]. A case function is identified by its PySMT abstract syntax tree.

Decision Diagrams (DDs): DDs or arithmetic circuits are used to compactly represent logical formulas by compiling them into directed acyclic graphs (DAGs) whose internal nodes are labeled by atoms and whose leaf nodes are labeled by expressions. Every internal node has two edges labeled with *true* ($\top$) and *false* ($\bot$) (traditionally called *high* and *low*) that correspond to the atom of the internal node being satisfied or not. We denote the label of an internal node f by f_C or $C(f)$, its two child nodes as $f_\top$ and $f_\bot$, following the high and low edge respectively, the label of an edge e as $v(e)$ and the expression of a leaf node l as $exp(l)$.

Binary decision diagrams (BDDs) use Boolean variables as atoms and truth values ($\{\top, \bot\}$ or $\{1, 0\}$) as expressions. Any propositional logical formula can be compiled into a BDD and decision diagrams support various efficient operations. E.g., computing SAT for a formula represented by a BDD corresponds to determining if there is a path from the root to 1. We consider three extensions of BDDs: LDDs, ADDs and XADDs (Table 1). LDDs support LRA atoms over continuous variables in internal nodes while ADDs allow an arbitrary number of leaf expressions labeled with real numbers. XADDs extends both by allowing LRA atoms in internal nodes as well as arbitrary symbolic polynomial functions over continuous variables in leaf nodes. Some XADD operations restrict the leaf nodes to be, e.g., linear functions.

XADDs [Sanner *et al.*, 2011] represent case functions with linear inequality conditions and polynomial values. Every path in an XADD corresponds to a case and its DAG structure allows compact representation of multiple cases with common structure (see Fig. 1 for an example XADD). For a path $p =$

BDD	**LDD**	**ADD**	**XADD**
$\mathbb{B}^n \mapsto \mathbb{B}$	$\mathbb{B}^n \times \mathbb{R}^m \mapsto \mathbb{B}$	$\mathbb{B}^n \mapsto \mathbb{R}$	$\mathbb{B}^n \times \mathbb{R}^m \mapsto \mathbb{R}$
Boolean	Boolean + LRA	Boolean	Boolean + LRA
$\{0, 1\}$	$\{0, 1\}$	$c \in \mathbb{R}$	Polynomial

Table 1: Overview of decision diagrams showing the type of labels for internal nodes and leaf nodes. While BDDs, LDDs, and ADDs have *constants* as leaves, XADDs have *symbolic polynomials* as leaves.

$n_1 \rightarrow_{e_1} \ldots \rightarrow_{e_{n-1}} n_n \rightarrow_{e_n} l$, the case condition and function can be obtained by taking the conjunction of the labels of the nodes and the expression of the leaf node l, respectively:

$$\theta(p) = \bigwedge\nolimits_{n_i \in p} ite(v(e_i), C(n_i), \neg C(n_i)), f(p) = exp(l)$$

One of the advantages of DDs is that they allow operations to be performed on them. The *if-then-else* (ite) construct allows DDs to be combined. E.g., given two decision diagrams d_1 and d_2 and a condition atom c, we can construct a new diagram $d = ite(c, d_1, d_2)$. The new diagram introduces a new internal node n with $n_C = c, n_\top = d_1$ and $n_\bot = d_2$, i.e., it behaves as d_1 if c is fulfilled and as d_2 otherwise. By using the so-called *apply* operation we can apply a binary operator to two decision diagrams. This operation constructs a new DD that corresponds to the composition of the two functions represented by the input diagrams. For BDDs and LDDs, diagrams are combined using $\wedge$ and $\vee$, for ADDs and XADDs, diagrams can also be combined using element-wise sum and product, and both support marginalization for Boolean variables. In addition, XADDs also support integration for real variables. We refer to marginalization through summation and integration (based on the variable type) as SO, short for sum-out. For real variable x, integer variable i, Boolean variable b and an XADD X_f corresponding to a polynomial case function f over x, i, b, we obtain $\text{SO}(x, X_f) = \int_{-\infty}^{\infty} f(x)dx$, $\text{SO}(i, X_f) = \sum_{i=-\infty}^{\infty} f$ and $\text{SO}(b, X_f) = \sum_{b \in \{\bot, \top\}} f$. We write $\text{SO}(b, \text{SO}(i, \text{SO}(x, X_f)))$ as $\text{SO}^*(\{x, i, b\}, X_f)$.

3 XADD Compilation for WMI

We propose a top-down compilation scheme to convert an SMT(LA) formula or polynomial case function from a nested arithmetic and ite abstract syntax tree to a valid XADD with a consistent node ordering. Our scheme recursively applies a set of grammar rules defined inductively in Table 2. The three base cases convert polynomials (including constants) to equivalent single-leaf XADDs and SMT(LA) literals a or $\neg a$ to 0-1 XADDs with a single test on a corresponding to $ite(a, 1, 0)$ or $ite(a, 0, 1)$, respectively. For composite expressions, the terms are recursively compiled and combined using the XADD *apply* operator [Sanner *et al.*, 2011] that automatically collapses paths sharing sub-diagrams.

Example 1. *The XADD depicted left in Fig. 1 is obtained by compiling the WMI problem with* $\theta = x \geq v \wedge (x < y \vee x \leq z)$ *and* $w = ite(x \leq z \wedge x \geq w, 3x^2, 2x)$.

Marginalization for XADDs is currently supported for tests over Boolean variables, LRA atoms and difference arithmetic atoms, i.e., over a subset of LIA. In this paper we consider

Expression e	**Compiled expression** $comp(e)$
val	$leaf(val)$
a	$ite(a, 1, 0)$
$\neg a$	$ite(a, 0, 1)$
$\theta_1 \wedge \theta_2$	$apply(\wedge, comp(\theta_1), comp(\theta_2))$
$\theta_1 \vee \theta_2$	$apply(\vee, comp(\theta_1), comp(\theta_2))$
$\neg(\theta_1 \wedge \theta_2)$	$apply(\vee, comp(\neg\theta_1), comp(\neg\theta_2))$
$\neg(\theta_1 \vee \theta_2)$	$apply(\wedge, comp(\neg\theta_1), comp(\neg\theta_2))$
$f_1 + f_2$	$apply(+, comp(f_1), comp(f_2))$
$f_1 * f_2$	$apply(*, comp(f_1), comp(f_2))$
$ite(\theta, f_1, f_2)$	$apply(+, apply(*, comp(\theta), comp(f_1)),$ $apply(*, comp(\neg\theta), comp(f_2)))$

Table 2: Compilation rules for XADDs using a for SMT(LA) atoms, i.e., Boolean variables or inequalities, θ for arbitrary SMT(LA) formulas, val for polynomial functions and f for polynomial case functions.

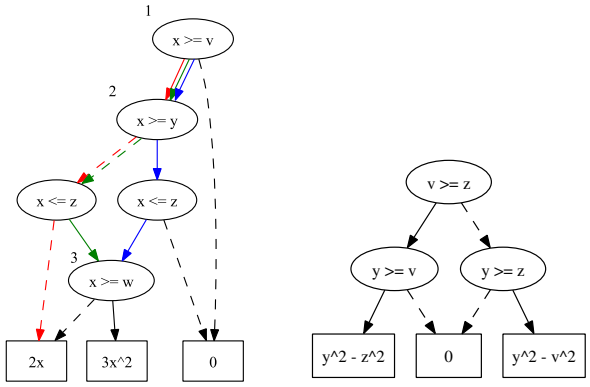

Figure 1: Illustration of two XADDs. The colors highlight different paths used in examples and are not part of the XADD data structure. Integrating out x from the red path results in the right XADD.

marginalization through summation and integration, leaving other operations, such as maximization, as future work.

Theorem 1. *Given an SMT(LA) formula* θ *and a polynomial case function* w *both over Boolean, integer and real variables* A, Z *and* $\mathbf{x}$*, then, using* $X_\psi = comp(\psi)$*:*

$$WMI(\theta, w|A, Z, \mathbf{x}) = exp(\text{SO}^*(A \cup Z \cup \mathbf{x}, X_\theta * X_w)).$$

Proof. Any case function defined by the input language of nested ite and arithmetic combinations can be compiled to an XADD by recursively applying the rules of Table 2. Similarly, every SMT(LA) function can be compiled to a 0-1 case function by the same procedure. Since $X_\theta * X_w = X_{ite(\theta,w,0)}$ and $WMI(\theta, w) = WMI(\top, ite(\theta, w, 0))$, summing out all variables from $X_\theta * X_w$ yields the weighted model integral. □

4 Novel XADD Partial Integrator

Having clarified that WMI-amenable theories of interest can be compiled to XADDs, we now discuss the WMI computation.

The computational cost of the XADD marginalization (SO) algorithm introduced by Sanner et al. (2012) is often prohibitive even for relatively simple models. We identify the main issue as the inability to effectively cache intermediate results of the integration process in the XADDs DAG and in this

section we aim to develop more efficient alternatives. In the following, we: 1) review Sanner et al.'s algorithm that is based on path enumeration; 2) introduce and discuss two variants of a novel algorithm based on bound resolution that introduce caching to SO; 3) conclude with an extension of XADDs and the marginalization algorithms to integer theories.

Every internal node containing a numeric variable x imposes an upper-bound on one child and a lower bound on the other. For the sake of brevity, we assume in our algorithms and discussions that upper bounds are always imposed on the high child and that all numeric variables are continuous (not integer). In practice, the algorithm would detect if an internal node test imposes a lower or upper bound and swap the child nodes if necessary. Note that w.r.t. integrating continuous variables strict and non-strict inequalities are equivalent.

Path Enumeration: Sanner et al. 's SO algorithm is summarized in Alg. 1. Essentially, it works like a symbolic version of DPLL, expanding the XADD DAG into an XADD tree, i.e., it recursively traverses every path in the XADD, collecting all expressions that can bound the integration variable x along the way. Reaching a leaf, the recursion returns the leaf integral $\int_{x \in C} f(x)dx$, where the set C is the intersection of bounds on x collected during the descent. Since C may depend parametrically on all non-integrated variables, the integral is computed symbolically as follows. For each bound pair $u \in U, l \in L$, we consider the case where u is the smallest of all possible upper bounds and l is the largest of all lower bounds, described by the formula $\theta_{ul} := \bigwedge_{u' \in U}(u \leq u') \bigwedge_{l' \in L}(l \geq l') \wedge (u \geq l)$, where the last atom ensures consistency (the upper bound must be at least as large as the lower bound for the integral to be well-defined). These formulas partition the space completely, so the integral of $\{\theta : f\}$ is then the case function $\{\theta_{ul} : \int_{x=l}^{u} f(x)dx \mid ul \in U \times L\}$. Let us quickly illustrate this with an example.

Example 2. *Consider the case function corresponding to the red path of Fig. 1 (omitting the cases where the result is* 0*):*

$$f = \{x \geq v \wedge x < y \wedge x > z : 2x\}; \textit{then}$$

$$\int_{-\infty}^{\infty} f dx = \{y \geq v \geq z : y^2 - z^2, y \geq z > v : y^2 - v^2\}$$

The XADD of that integral is illustrated in Fig. 1, right.

The computational disadvantage of this algorithm is that it does not exploit the DAG structure of the XADD. This is reflected in the fact that the SO function receives all upper/lower bounds accumulated so far as an argument, hence the output of each recursive call depends on the entire context from which it is called. This eliminates any opportunity for caching, and what looks as dynamic programming is essentially a tree-structured recursion which must always traverse exponentially many paths. This renders the algorithm inapplicable on anything but toy models.

This is also akin to the depth-first PRAiSe algorithm [Braz *et al.*, 2016] which explicitly uses a symbolic extension of DPLL but not XADDs to represent piecewise functions.

Bound Resolution: We will now develop two alternatives of this algorithm that introduce increasing amounts of caching. The main idea is as follows: instead of collecting all candidate bounds until the bottom of the recursion and then introducing

Algorithm 1 Integration using path enumeration.

```
1: procedure SO(var x, XADD f, U, L)
2:                          ▷ U: upper-bounds, L: lower-bounds
3:   if f is terminal then
4:     return integrate(x, f, U, L)
5:   if x ∈ f_C then
6:     (l, u) ← get bounds(x, f_C)
7:     return SO(x, f_⊤, U ∪ {u}, L)
                 +SO(x, f_⊥, U, L ∪ {l})
8:   return ite(f_C, SO(x, f_⊤, U, L), SO(x, f_⊥, U, L))
```

cases to compare among them, we maintain a single upper and lower bound candidate. As soon as a new potential upper or lower bound appears due to some test, the recursion introduces two cases: 1) if the new upper bound is greater than the current one, we keep the current bound and descend on child node; 2) if the new upper bound is less than the current one, it replaces the current bound before we descend on the child node. In the latter case, a consistency test is added to enforce that the new upper bound is greater than the current lower bound. For our first version, Alg. 2, we end up with the following recursive rule for the upper bound:

$$f_u \leftarrow ite(u_{new} > u, \mathrm{SO}(x, f_\top, u, l), \\ (u_{new} \geq l) * \mathrm{SO}(x, f_\top, u_{new}, l)) \ .$$

The output of this algorithm is identical to the output of Alg. 1, however, the context influence is reduced to only the upper and lower bound candidate. Hence, the triple u, l, f can serve as the key for caching the result.

Example 3. *Reconsider the left XADD from Fig. 1. Both the green path and the blue path will at some point encounter the call* $\mathrm{SO}(x, 3, z, v)$ *as on both paths* z *and* v *appear as upper- and lower bound to* x*, respectively, and both paths go through node* 3*. However, since the two paths go through different branches of node* 2*, the recursion for green also invokes* $\mathrm{SO}(x, 3, v, y)$*, whereas the recursion for blue invokes* $\mathrm{SO}(x, 3, y, z)$*. The latter two calls cannot profit from caching even though the paths generating them rejoin in the diagram.*

Finally, we consider the version of Alg. 2b which is completely context-independent. The key idea is that instead of calling $\mathrm{SO}(x, f_\top, u_{new}, l)$, the same result is obtained by calling $\mathrm{SO}(x, f_\top, u_S, l_S)$, where u_S and l_S are symbolic variables, and then substituting u_S by u_{new} and l_S by l in the resulting XADD. Similarly, we obtain $\mathrm{SO}(x, f_\top, u, l))$ by substituting u_S by u and l_S by l. Notice that while u_{new} is available at the time of the recursion call, as it comes from the current node, we want to eliminate u from the list of arguments. Hence, the second branch of the ite expression is left with the dummy variable u_S as a placeholder. This will be substituted with the various upper bounds as the algorithm emerges from the recursion. We end up with the following recursive step:

$$f_u \leftarrow ite(u_{new} > u_S, \mathrm{SO}(x, f_\top), \\ (u_{new} \geq l_S) * \mathrm{SO}(x, f_\top)\{u_S / u_{new}\}) \ ,$$

where u_S/u_{new} denotes the substitution of the dummy variable u_S by the concrete bound u_{new}. We have now eliminated

Algorithm 2 Integration using bound resolution

(2a) Algorithm for bound-pair caching

1: **procedure** SO(var x, XADD f, u, l)
2: ▷ u: current upper-bound, l: current lower-bound
3: **if** f is terminal **then**
4: **return** $\text{integrate}(x, u, l, f)$
5: **if** x $\notin f_C$ **then**
6: **return** $ite(f_C, \text{SO}(x, f_\top, u, l), \text{SO}(x, f_\bot, u, l))$
7: $(l_{new}, u_{new}) \leftarrow \text{get bounds}(x, f_C)$
8: $f_u \leftarrow ite(u_{new} > u, \text{SO}(x, f_\top, u, l), (u_{new} \geq l) * \text{SO}(x, f_\top, u_{new}, l))$
9: $f_l \leftarrow ite(l_{new} < l, \text{SO}(x, f_\bot, u, l), (l_{new} \leq u) * \text{SO}(x, f_\bot, u, l_{new}))$
10: **return** $f_u + f_l$

(2b) Algorithm for symbolic caching

1: **procedure** SO(var x, XADD f)
2:
3: **if** f is terminal **then**
4: **return** $\text{integrate}(x, f)$
5: **if** x $\notin f_C$ **then**
6: **return** $ite(f_C, \text{SO}(x, f_\top), \text{SO}(x, f_\bot))$
7: $(l_{new}, u_{new}) \leftarrow \text{get bounds}(x, f_C)$
8: $f_u \leftarrow ite(u_{new} > u_S, \text{SO}(x, f_\top), (u_{new} \geq l_S) * \text{SO}(x, f_\top)\{u_S/u_{new}\})$
9: $f_l \leftarrow ite(l_{new} < l_S, \text{SO}(x, f_\bot), (l_{new} \leq u_S) * \text{SO}(x, f_\bot)\{l_S/l_{new}\})$
10: **return** $f_u + f_l$

any context information from the call to SO, and the result of $\text{SO}(x, f)$ depends only on f. This algorithm is able to reuse *any partial computation*, exactly as the marginalization algorithm on ADDs. Moreover, it can be implemented in bottom-up message-passing style, as is characteristic for dynamic programming algorithms. To see the difference with the previous version, note that the entire computation of the integral under node 3 on Fig. 1 can be reused for both green and blue paths.

It is important to note that the second version of the algorithm does not subsume the first. Unlike the Boolean setting, where more caching almost universally improves performance, for linear theories this is not always the case. Having concrete bounds allows us to prune the diagram early by detecting conflicting bounds, e.g., if $u_{new} \geq l$ can be detected to be unsatisfiable, the recursion can be terminated. In practice, the first version seems to be the most versatile.

Theory-specific Considerations: We currently support XADDs that, aside from Boolean variables, can contain either real or integer variables. For integer variables symbolic summation is used instead of symbolic integration.[1] In order to guarantee that (symbolic) summation bounds are integral, we need to restrict the language of the tests. One amenable sub-theory of LIA is difference arithmetic. Additionally, for the integration of real variables, strict and weak inequalities are equivalent (the set of boundary points has measure zero), hence negation is obtained by simply flipping the inequality. However, for integer variable this is not the case, e.g., the inverse of $x \leq 5$ is $x > 5$ or $x \geq 6$, which leads to some implementation differences. Finally, a pruning routine for inconsistent paths in XADDs is implemented using linear programming for real variables, while for integer variables we currently use SMT(LIA) solvers as an oracle for testing inconsistency, as well as linear programming for a relaxed version of inconsistency.

5 XADD Integration for WMI

We can now show how WMI can be realized using XADD based SO-algorithms. Specifically, we introduce two techniques for solving single WMI problems and then propose a partial-WMI algorithm for computing probabilities of sets of WMI queries efficiently.

WMI using Repeated SO: Calculating the WMI corresponds to summing out a set of variables from an XADD, i.e., $\text{SO}^*(vars, X)$. The simplest way to compute the outcome is to reuse an SO algorithm for XADDs, e.g., path enumeration or bound resolution, and sum out variables one by one.

WMI using Mass-SO: Alternatively, we can tweak our recursive bound-resolution algorithm to immediately sum-out all variables before emerging. This algorithm requires an *variable-ordered* XADD in which the ordering of tests in the XADD depends on the variables occurring in them. Specifically, given a variable ordering $O = v_1, v_2, ..., v_n$ we call an XADD variable-ordered with respect to O if any test whose *last* variable has rank i, i.e., it has position i in the ordering, occurs deeper in the diagram than any test whose last variable has a lower rank $j < i$. The last variable of a test is the variable in the test with the highest rank. The modified algorithm now proceeds by first summing out the variable v_1 with the lowest rank using the original bounds-resolve algorithm (Alg. 2) and, whenever a test is encountered whose last variable v_j is ranked higher than the current variable v_i, summing out variables $v_{i+1}, ..., v_{j-1}, v_j$ recursively before proceeding to sum out v_i from the resulting diagram. The ordering guarantees that after summing out a variable with rank i, the resulting diagram contains no variable with rank $j \geq i$.

Partial WMI for Computing Query Probabilities: In practice, weighted model integration is used to compute query probabilities, where queries are arbitrary SMT(LA) formulas over both Boolean and numeric variables. Given a query q over variables $vars(q) \subseteq D$, its probability is obtained by computing the ratio $\frac{WMI(\theta \wedge q, w|D)}{WMI(\theta, w|D)}$. Current solvers obtain this answer by computing both volumes separately and dividing the results. For a set Q of n queries, this requires $n + 1$ WMI computations. By exploiting symbolic SO, we propose a new technique for computing probabilities for sets of queries. This approach can drastically reduce the execution time when the number of variables D_Q occurring in any query is much lower than the number of variables in the domain. The algorithm WMI^* consists of three steps: 1) it pre-computes a temporary diagram X_Q by summing out variables $D \setminus D_Q$

[1] Implemented in Python using *sympy* [Meurer et al., 2017].

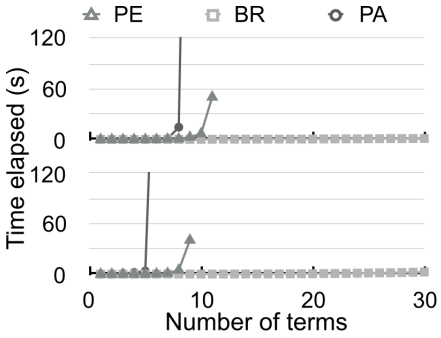

Figure 2: The execution time (top: mutual exclusivity, bottom: XOR) grows much slower for our algorithm (BR) than for path enumeration (PE) and the WMI solver (PA) both of which time out for larger numbers of terms.

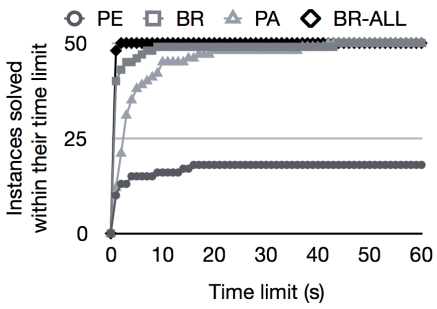

Figure 3: Both SO and mass-SO using bound resolution (BR, BR-ALL) are competitive with the WMI solver (PA) on the number of problems solved below increasing time limits. The path enumeration approach (PE) fails to solve several problems within $60s$.

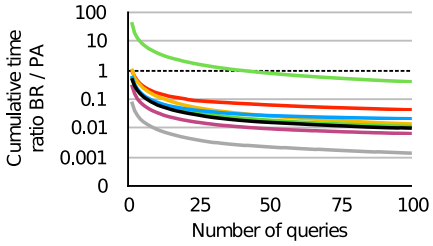

Figure 4: The ratios between the cumulative execution times to solve a number of queries using our bound resolution algorithm (BR) and the WMI solver (PA) decrease to below 1, even if the initial queries were slower to compute. Our algorithm timed out while computing the partial WMI for problems 4 and 7.

that do not occur in any query; 2) it computes $WMI(\theta, w)$ by summing out variables V_Q from X_Q; and 3) for every query q it computes $WMI(\theta \wedge q, w)$ by summing out variables V_Q from $X_Q * X_q$.

Theorem 2. *Given θ, Q and w over variables D, as before, then WMI^* computes $\{\frac{WMI(\theta \wedge q, w|D)}{WMI(\theta, w|D)} | \forall q \in Q\}$.*

Proof. The algorithm computes $WMI(\theta \wedge q, w|D)$ and $WMI(\theta, w|D)$, where $q = \top$, using SO. Since $WMI(\theta \wedge q, w|D) = \text{SO}^*(D, X_\theta * X_q * X_w)$ and X_q is independent of $D \setminus D_Q$ it follows that the WMI can be computed as $\text{SO}^*(D_Q, \text{SO}^*(D \setminus D_Q, X_\theta * X_w) * X_q)$. □

X_Q is the result of partial WMI, i.e., $WMI(\theta, w|D \setminus D_Q)$, and it should be clear that this can also be used to compute, e.g., $\text{argmax}_{D_Q} X_Q$, which is a standard XADD operation.

6 Empirical Evaluations

We analyze the performance of our improved SO algorithm and how XADDs can be used to tackle WMI problems. First, we demonstrate our algorithms ability to exploit caching in very structured diagrams. Second, we compare our algorithm to the current XADD-based SO algorithm [Sanner and Abbasnejad, 2012] and the state-of-the-art WMI solver [Morettin *et al.*, 2017] (PA) on WMI computation for synthetic benchmark problems from the paper by Morettin et al. Third, we show how partial integration for XADDs can be leveraged to repeatedly compute query probabilities and compare this approach to the PA solver. We generally use the ordering in which literals (or variables for mass-SO) appear during compilation. The code for the implementations used in our evaluation can be found at: `https://github.com/xadd-wmi/xadd-wmi.github.io`.

Structured Diagrams: Our SO algorithm exploits caching in order to avoid redundant computations when sub-diagrams can be reached by multiple paths. We investigate the performance of our algorithm on two common types of heavily structured problems: 1) mutual-exclusivity (also called *exactly one*) formulas; and 2) XOR formulas.

These structured constraints arise naturally in data and in modeling multi-valued attributes using arithmetic circuits, where each multi-valued attribute is converted into multiple binary variables with mutual exclusivity constraints.

For both problem types, given a number of terms n, we introduce $n+1$ variables $V = \{c_1, ..., c_n, x\}$, $n+1$ terms that introduce real-valued bounds for the variables (e.g., $b_x = ((l_x \leq x) \wedge (x \leq u_x))$ where $l_x, u_x \in \mathbb{R} \wedge (l_x < u_x)$) and n terms $T = \{x \leq c_1, ..., x \leq c_n\}$. Then we compute the mutual-exclusivity formula f_{ME} and the XOR formula f_{XOR} as follows:

$$f_{ME} = (\textstyle\bigwedge_{v \in V} b_v) \wedge \mathit{exactly\_one}((x \leq c_1), ..., (x \leq c_n))$$
$$f_{XOR} = (\textstyle\bigwedge_{v \in V} b_v) \wedge ((x \leq c_1) \veebar ... \veebar (x \leq c_n)).$$

We compute $WMI(f_{XOR}, 1|V)$ and $WMI(f_{ME}, 1|V)$ for increasing n using bound resolution, path enumeration, and the state-of-the-art WMI solver.

For both the XOR- and mutual-exclusivity-problems, our algorithm is able to compute the volume in well under 120 seconds as its execution time grows slowly with the problem size. Therefore, it outperforms the state-of-the-art approaches that both exhibit exponentially growing execution times, see Fig. 2.

Predicate-abstraction Solver Benchmark: In the paper [Morettin *et al.*, 2017], the authors demonstrate that their approach outperforms Praise [Braz *et al.*, 2016] and block-clause based approaches [Belle *et al.*, 2015] using a set of synthetically generated problems. Using the publicly available code for the PA solver we generated 50 such WMI problems and compared the performance of: 1) SO using path enumeration; 2) SO using bound-resolution; 3) mass-SO using bound-resolution; and 4) the PA solver. For every problem we record the execution time per solver, with a time-out of $60s$.

Both bound-resolution based algorithms and the PA solver can compute any of the problems within 60 seconds, while the path enumeration based algorithm times out for many of the problems (Fig 3). Both bound-resolution based algorithms outperform the predicate-abstraction based solver by solving more problems faster. Additionally, the mass-SO approach solves all but one problem within a second.

Answering Sets of WMI queries: As described earlier, computing the probabilities of a set of queries can be sped

up using XADDs. We compare our approach to the state-of-the-art WMI solver by comparing the time required to solve an increasing number of queries for a set of benchmark WMI problems consisting of random pairs of nested SMT formulas and polynomial case functions in the form of PySMT ASTs, generated using the predicate-abstraction solver software package. For every problem 100 queries were generated consisting of different inequalities over the variable of interest.

Our experiments show that our partial integration approach can achieve large speedups as the number of queries increases. Since only few variables occur in the queries, our approach results in much lower execution times per query as it avoids repeatedly computing the WMI for all variables. Time-outs can occur since, for arbitrary XADDs, operations may lead to exponential numbers of nodes, however, in practice variable-reordering strategies can help alleviate this problem. Reordering schemes based on sifting have already shown to be effective for arithmetic based diagrams [Chaki *et al.*, 2009].

7 Conclusion

Weighted model integration (WMI) is a promising and general approach to reasoning about mixed discrete and continuous spaces needed for probabilistic inference. We proposed the first XADD-based WMI solver that can cache partial computations, smartly exploit DAG structure in integration, and handle integer-valued variables. Empirical results show this new WMI solver performs comparable to state-of-the-art solvers and often drastically better. We also showed that XADDs are a suitable compilation language for WMI and we hope it serves as a step towards making WMI as versatile a tool as WMC.

References

[Albarghouthi *et al.*, 2017] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Quantifying program bias. *CoRR*, abs/1702.05437, 2017.

[Bacchus *et al.*, 2009] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Solving #sat and bayesian inference with backtracking search. *JAIR*, 34:391–442, 2009.

[Belle *et al.*, 2015] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of IJCAI*, pages 2770–2776, 2015.

[Belle *et al.*, 2016] Vaishak Belle, Guy Van den Broeck, and Andrea Passerini. Component caching in hybrid domains with piecewise polynomial densities. In *Proc. of AAAI*, pages 3369–3375, 2016.

[Belle, 2017] Vaishak Belle. Weighted model counting with function symbols. In *Proc. of UAI*, 2017.

[Braz *et al.*, 2016] Rodrigo de Salvo Braz, Ciaran O'Reilly, Vibhav Gogate, and Rina Dechter. Probabilistic inference modulo theories. In *Proc. of IJCAI*, pages 3591–3599, 2016.

[Chaki *et al.*, 2009] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 53–60. IEEE, 2009.

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

[Chistikov *et al.*, 2015] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. In *TACAS*. 2015.

[Choi *et al.*, 2013] Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *Proceedings of 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*, pages 121–132. Springer, 2013.

[Domshlak and Hoffmann, 2007] Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007.

[Fierens *et al.*, 2013] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Journal of Theory and Practice of Logic Programming*, 15:358–401, 2013.

[Gario and Micheli, 2015] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 373–384, 2015.

[Meurer et al., 2017] Aaron Meurer et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[Morettin *et al.*, 2017] Paolo Morettin, Andrea Passerini, and Roberto Sebastiani. Efficient weighted model integration via smt-based predicate abstraction. In *Proc. of IJCAI*, pages 720–728, 2017.

[Sanner and Abbasnejad, 2012] Scott Sanner and Ehsan Abbasnejad. Symbolic variable elimination for discrete and continuous graphical models. In *Proc. of AAAI*, 2012.

[Sanner *et al.*, 2011] Scott Sanner, Karina Valdivia Delgado, and Leliane Nunes De Barros. Symbolic dynamic programming for discrete and continuous state mdps. In *Proc. of UAI*, 2011.

[Suciu *et al.*, 2011] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.